GUIDED SYNTHESIS OF NETWORK BEHAVIOR

BY

WENXUAN ZHOU

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Doctoral Committee:

Associate Professor Matthew Caesar, Chair & Director of Research
Associate Professor Nikita Borisov
Associate Professor P. Brighten Godfrey
Professor Jennifer Rexford, Princeton University

Abstract

In the past decades, the world has witnessed how essential modern networks, such as data centers and enterprise networks, have become in our daily lives. However, on the other hand, configuring and maintaining a modern network, is a challenging and error-prone process. Administrators must often consider security policies from a variety of sources simultaneously, including regulatory requirements, industry standards, to mitigate attack vectors. Erroneous implementation of a policy, however, can result in costly data breaches and intrusions. Relying on humans to discover and troubleshoot violations is slow and prone to error, considering the speed at which new attack vectors propagate and the increasing network dynamics, partly an effect of SDN. To ensure the network is always in a state consistent with the desired policies, administrators need frameworks to automatically diagnose and repair violations as the network evolves.

To address this problem, in this dissertation, we present a system automatically synthesizing network changes that meets a network correctness specification given as a policy. If we consider a network as a distributed program, the problem here is essentially a program synthesis problem. Recent work on program synthesis illustrates many benefits of allowing the user to augment the correctness specification with some guidance. We adopt a similar philosophy: our system is guided by user instructions to constrain the space of allowed implementations in order to keep pace with network dynamics. As the foundation of our system, we first develop a verification technique that detects network-wide invariant violations responsively. Based on the verification results, our core algorithm repairs network updates in two aspects. If an update violates a policy defined by an administrator, such as reachability or segmentation, our algorithm transforms the update into one that complies with the policy. In addition, given two correct network states, our algorithm synthesizes a feasible and efficient update ordering to migrate the network from one to the other.

With our prototype implementation, we tested our system on physical testbed, emulated SDN

networks, and a large enterprise network's operational traces. We demonstrated that it is practical and efficient to use user instructions as guidance to incrementally build/maintain a network state, where desirable properties are automatically preserved all the time.

To my family.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my advisor, Professor Matthew Caesar. For more than seven years, he has been an inspiring and reliable force guiding me towards "synthesizing" not only my thesis work but also the confidence and the ability to discover and conduct important research. I also thank him for unconditional and persistent support as a friend. When I came to the United States the first time seven years ago, it was he who made adapting extremely easy and made me feel like UIUC is where I belonged.

A second person I am greatly indebted to is Professor Brighten Godfrey. Meetings with him throughout all these years have taught me numerous brilliant technical insights, presentation skills, and the ability to develop raw ideas into mature research. The projects advised by both him and Prof. Matthew Caesar laid the foundation of my thesis.

I have been fortunate to have an amazing set of collaborators. The work in Chapter 2 of this dissertation was joint with Ahmed Khurshid; Chapter 3 was with Jin Dong and Jason Croft; and Chapter 4 was with Jason Croft, Bingzhe Liu, and Elaine Ang. During my graduate studies, I also very much enjoyed and learned from my collaborations on work not included in this dissertation, with Hitesh Ballani, Nikita Borisov, David Choffnes, Peter Druschel, Paul Francis, Amir Houmansadr, Stevens Le Blond, and Qingxi Li.

I want to thank my Ph.D. committee members Professor Nikita Borisov and Professor Jennifer Rexford for their valuable comments and validation in various stages of my thesis work. I also want to thank all of my friends, friends back at China and new friends I met here, for their help and believing in me. Forgive me not to mention all the names, as it is almost impossible to do so. I would like to take this opportunity to thank the many staff members in the CS Department for their tremendous help.

Finally, I am especially grateful to my mother, Jie Zhou, and my husband, Yan Zhai. My

mother never gave me any idea that I could not do whatever I wanted to do or be whomever I wanted to be. The home I was raised up in was filled with nothing but love, fun, tasty food and books. Now I am fortunate enough to have a small family like such with my husband, Yan. I owe an immeasurable debt to Yan, for being my best friend, and for giving me my ultimate inspiration. To them, I dedicate this dissertation.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern networks, such as data center networks and enterprise networks, are essential in our daily lives. Network operators often establish a set of correctness conditions to ensure successful operation of the network, such as the preference of one path over another, the prevention of untrusted traffic from entering a secure zone, or loop and black hole avoidance. As networks become an increasingly crucial backbone for critical services, the ability to construct networks that obey correctness criteria is becoming even more important.

On the other hand, packet forwarding in modern networks is a complex process, involving codependent functions running on hundreds or thousands of devices, such as routers, switches, and firewalls from different vendors. Although a substantial amount of effort is required to ensure networks' correctness, security and fault tolerance, faults in the network state still arise commonly in practice, including loops, suboptimal routing, black holes and access control violations that make services unavailable or prone to attacks (e.g., DDoS attacks), or result in costly data breaches. How can a human reason about the trillions of possibilities? The answer is no, people cannot comprehend modern networks themselves.

Then can we rely on computers to solve this problem? Many have tried. The challenge of constructing networks that work correctly has led to a variety of techniques to verify network behaviors [16, 17, 40, 47], as well as work on configuration analysis [26, 71], and symbolic execution [22]. Those techniques are useful to find out when things go wrong, but they operate offline, and thus only find bugs after they happen. Researchers have also investigated automated synthesis techniques to derive network states from a set of pre-specified correctness conditions [60, 64, 67]. However, these tools are designed for uses similar to a compiler, and therefore do not work in cooperation with network users (either operators or existing applications). For the same reason, they would not be feasible to interactively synthesize network changes. How can a computer know what a network is supposed to do? Like humans, computers alone cannot comprehend the networks.

Therefore, instilling correctness into modern networks is a task that neither computers nor humans alone can solve. In fact, problems like this also occur in domains outside of networking, for example, theorem proving where billions of alternatives could exist for a non-trivial problem, or missile guidance and aircraft control where humans cannot react fast enough but computers don't know what needs to be done. In all of these situations, we need to solve a strong AI problem. Guidance is the key of the solution approach, which is allowing a human to "guide" what a computer does. In theorem proving, a human gives hints and feedback to a solver, and in aircraft control, a human gives course directions which are corrected by the computer. These systems define feedback loops between humans and computers.

Thus, we argue the need for guided techniques to ensure correctness in modern networks. In particular, this thesis asks a question: *Is it possible to efficiently perform guided synthesis of network state?* To address this question, we designed and implemented a framework to synthesis network state in a manner that is guided by human intent.

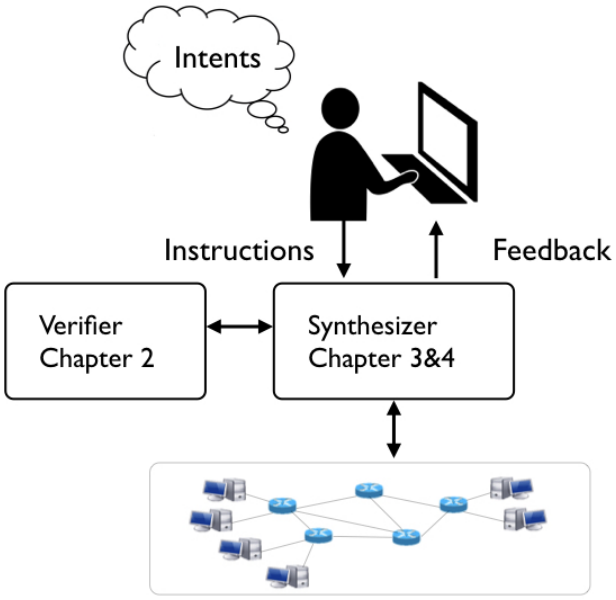


Figure 1.1: *System Architecture.*

An overview of our system is shown in figure 1.1. It interacts with user by 1) taking user input in the form of network instructions (from network operators or applications), and 2) (optionally) sending feedback back in the form of suggested changes on the network instructions. On the interaction with networking devices front, the system sends synthesized updates down to the network

and reads state from it. The layer in the middle does the computation. Then the question is how to design the middle layer. There are two key objectives. First, computer needs to know if the goal is reached. To address this, we build a responsive network verification layer. Second, there are many candidates that could lead us towards the goal. The computer needs to know which one to choose. More specifically, what should the state be, and when should state updates be sent. The next subsection gives more details about each of these techniques.

1.1 Contributions

In this section, we describe the key contributions of this dissertation.

- **Responsive Network Verification Layer** (§ 2) As the foundation of the system, in this component, the objective is to find out when and where things go wrong in real time. We present a design, *VeriFlow*, which demonstrates that the goal of real-time verification of network-wide invariants is achievable. *VeriFlow* leverages software-defined networking (SDN) to obtain a picture of the network as it evolves by sitting as a layer between the SDN controller and the forwarding devices, and checks validity of invariants as each rule is inserted, modified or deleted. However, SDN alone does not make the problem easy. In order to ensure real-time response, *VeriFlow* introduces novel incremental algorithms to search for potential violation of key network invariants — for example, availability of a path to the destination, absence of forwarding loops, enforcement of access control policies, or isolation between virtual networks.

- **Update Timing and Ordering Synthesis** (§ 3) Next, we study a following-up problem in the space: given two correct consecutive configurations, how can we guarantee a general set of properties are preserved during transitions from one configuration to another? This has been referred as *network consistency* [63] problem. As an answer, we present *CCG*, a system that enforces customizable network consistency properties with high efficiency. We highlight the network uncertainty problem and its ramifications, and propose a network modeling technique correctly derives consistent outputs even in the presence of uncertainty. More

importantly, we reduce the synthesis problem to a verification problem. The core algorithm of *CCG* leverages the uncertainty-aware network model, and synthesizes a feasible network update plan (ordering and timing of control messages) via repeated verifications. That is, *CCG* repairs network update sequences. In addition to ensuring that there are no violations of consistency requirements, *CCG* also tries to maximize update parallelism, subject to the constraints imposed by the requirements. Through emulations and experiments on an SDN testbed, we show that *CCG* is capable of achieving a better consistency vs. efficiency trade-off than existing mechanisms.

- **Update Content Synthesis** (§ 4) We then asked this question: "*is it possible to efficiently enforce correctness policies on arbitrary data plane states?*". That is, we now relax the constraints (in *CCG*) on configurations being correct with regard to policies and only repairing intermediate states, but also repair erroneous configurations. We propose *NEAt*, a system that provides network administrators with a network analogue of a smartphone's autocorrect. As a transparent layer, *NEAt* repairs, in real-time, updates from an SDN controller that violate generic policies such as reachability, service-chaining, and segmentation detected by the verification engine. *NEAt* repairs the updates by adding or removing a minimal number of rules in order to comply with the policy by casting the problem as an optimization problem.

1.2 Organization

The rest of this dissertation is organized as follows. In Chapter § 2 we present our real-time network verification technique, as the foundation of the infrastructure. Using the verification technique as the means of guidance, we then present our approaches to incrementally synthesize network behavior. In particular, we organize them in a way that we first tackle the problem from time dimension (Chapter § 3), and then from space dimension (Chapter § 4). Chapter § 5 discusses the main relevant approaches available today for diagnosing and synthesizing network states. Finally, Chapter § 6 summarizes the dissertation, discusses limitations and presents directions for future work.

Chapter 2

Responsive Network Verification Layer

In this chapter, I present the design of our responsive network verification layer, *VeriFlow*. Packet forwarding in modern networks is a complex process, involving codependent functions running on hundreds or thousands of devices, such as routers, switches, and firewalls from different vendors. As a result, a substantial amount of effort is required to ensure networks' correctness, security and fault tolerance. However, faults in the network state arise commonly in practice, including loops, suboptimal routing, black holes and access control violations that make services unavailable or prone to attacks (e.g., DDoS attacks). Software-Defined Networking (SDN) promises to ease the development of network applications through logically-centralized network programmability via an open interface to the data plane, but bugs are likely to remain problematic since the complexity of software will increase. Moreover, SDN allows multiple applications or even multiple users to program the same physical network simultaneously, potentially resulting in conflicting rules that alter the intended behavior of one or more applications [65].

One solution is to rigorously check network software or configuration for bugs prior to deployment. Symbolic execution [22] can catch bugs through exploration of all possible code paths, but is usually not tractable for large software. Analysis of configuration files [26, 71] is useful, but cannot find bugs in router software, and must be designed for specific configuration languages and control protocols. Moreover, using these approaches, an operator who wants to ensure the network's correctness must have access to the software and configuration, which may be inconvenient in an SDN network where controllers can be operated by other parties [65]. Another approach is to statically analyze snapshots of the network-wide data-plane state [16, 17, 40, 47, 70]. However, these previous approaches operate offline, and thus only find bugs after they happen.

This chapter studies the question, *Is it possible to check network-wide correctness in real time as the network evolves?* If we can check each change to forwarding behavior before it takes effect, we

can raise alarms immediately, and even prevent bugs by blocking changes that violate important invariants. For example, we could prohibit changes that violate access control policies or cause forwarding loops.

However, prior to our solution techniques for checking networks are inadequate for this purpose as they operate on timescales of seconds to hours [17, 40, 47]. [1] Delaying updates for processing can harm consistency of network state, and increase reaction time of protocols with real-time requirements such as routing and fast failover; and processing a continuous stream of updates in a large network could introduce scaling challenges. Hence, we need some way to perform verification at very high speeds, i.e., within milliseconds. Moreover, checking network-wide properties requires obtaining a view of network-wide state.

VeriFlow leverages software-defined networking (SDN) to obtain a picture of the network as it evolves by sitting as a layer between the SDN controller and the forwarding devices, and checks validity of invariants as each rule is inserted, modified or deleted. However, SDN alone does not make the problem easy. In order to ensure real-time response, *VeriFlow* introduces novel incremental algorithms to search for potential violation of key network invariants — for example, availability of a path to the destination, absence of forwarding loops, enforcement of access control policies, or isolation between virtual networks.

Our prototype implementation supports both OpenFlow [51] version 1.1.0 and IP forwarding rules, with the exception that the current implementation does not support actions that modify packet headers. We microbenchmarked *VeriFlow* using a stream of updates from a simulated IP network, constructed with Rocketfuel [12] topology data and real BGP traces [15]. We also evaluated its overhead relative to the NOX controller [29] in an emulated OpenFlow network using Mininet [7]. We find that *VeriFlow* is able to verify network-wide invariants within hundreds of microseconds as new rules are introduced into the network. *VeriFlow*'s verification phase has little impact on network performance and inflates TCP connection setup latency by a manageable amount, around 15.5% on average.

We give an overview of data plane verification and SDN (§ 2.1) before presenting *VeriFlow*'s design (§ 2.2), implementation (§ 2.3), and evaluation (§ 2.4). We then discuss future (§ 6.2) and

[1] The average run time of reachability tests in [40] is 13 seconds, and it takes a few hundred seconds to perform reachability checks in Anteater [47].

related work (§ 5.1), and conclude (§ 2.5).

2.1 Overview of Approach

VeriFlow adopts the approach of *data plane verification*. As argued in [47], verifying network correctness in the data plane offers several advantages over verifying higher-level code such as configuration files. First, it is closely tied to the network's actual behavior, so that it can catch bugs that other tools miss. For example, configuration analysis [26, 71] cannot find bugs that occur in router software. Second, since data-plane state has relatively simple formats and semantics that are common across many higher-layer protocols and implementations, it simplifies rigorous analysis of a network.

Early data plane verification algorithms were developed in [70], and systems include FlowChecker [16], Anteater [47], and Header Space Analysis [40]. The latter two systems were applied to operational networks and uncovered multiple real-world bugs, validating the data plane analysis approach. However, as noted previously, these are offline rather than real-time systems.

VeriFlow performs real-time data plane verification in the context of software defined networks (SDNs). An SDN comprises, at a high level, (1) a standardized and open interface to read and write the data plane of network devices such as switches and routers; (2) a *controller*, a logically centralized device that can run custom code and is responsible for transmitting commands (forwarding rules) to network devices.

SDNs are a good match for data plane verification. First, a standardized data plane interface such as OpenFlow [11] simplifies unified analysis across all network devices. Second, SDNs ease *real-time* data plane verification since the stream of updates to the network is observable at the controller.

SDN thus simplifies *VeriFlow*'s design. Moreover, we believe SDNs can benefit significantly from *VeriFlow*'s data plane verification layer: the network operator can verify that the network's forwarding behavior is correct, without needing to inspect (or trust) relatively complex controller code, which may be developed by parties outside the network operator's control.

2.2 Design

Checking network-wide invariants in the presence of complex forwarding elements can be a hard problem. For example, packet filters alone make reachability checks NP-Complete [47]. Aiming to perform these checks in real-time is therefore challenging. Our design tackles this problem as follows. First, we monitor all the network update events in a live network as they are generated by network control applications, the devices, or the network operator. Second, we confine our verification activities to only those parts of the network whose actions may be influenced by a new update. Third, rather than checking invariants with a general-purpose tool such as a SAT or BDD solver as in [17, 47] (which are generally too slow), we use a custom algorithm. We now discuss each of these design decisions in detail.

VeriFlow's first job is to track every forwarding-state change event. For example, in an SDN such as OpenFlow [51], a centralized controller issues forwarding rules to the network devices to handle flows initiated by users. *VeriFlow* must intercept all these rules and verify them before they reach the network. To achieve this goal, *VeriFlow* is implemented as a shim layer between the controller and the network, and monitors all communication in either direction.

For every rule insertion/deletion message, *VeriFlow* must verify the effect of the rule on the network at very high speed. *VeriFlow* cannot leverage techniques used by past work [16, 40, 47], because these operate at timescales of seconds to hours. Unlike previous solutions, we do not want to check the entire network on each change. We solve this problem in three steps. First, using the new rule and any overlapping existing rules, we slice the network into a set of *equivalence classes* (ECs) of packets (§ 2.2.1). Each EC is a set of packets that experience the same forwarding actions throughout the network. Intuitively, each change to the network will typically only affect a very small number of ECs (see § 2.4.1). Therefore, we find the set of ECs whose operation could be altered by a rule, and verify network invariants only within those classes. Second, *VeriFlow* builds individual *forwarding graphs* for every modified EC, representing the network's forwarding behavior (§ 2.2.2). Third, *VeriFlow* traverses these graphs (or runs custom user-defined code) to determine the status of one or more invariants (§ 2.2.3). The following subsections describe these steps in detail. Figure 2.1 shows the placement and operations of *VeriFlow* in an SDN.

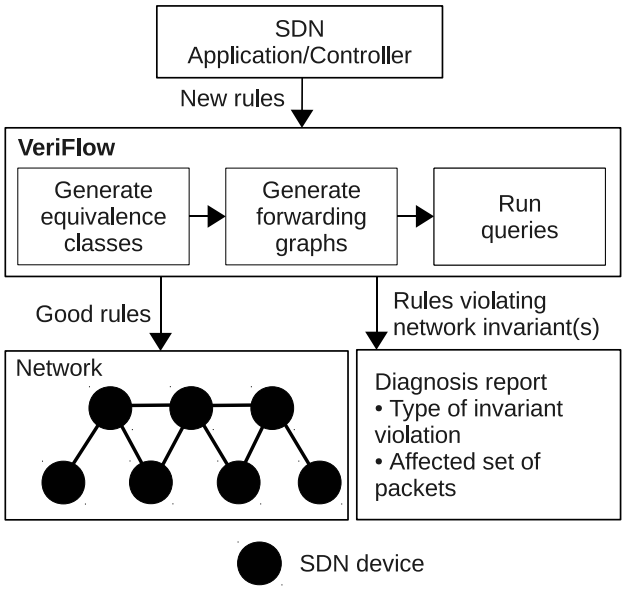


Figure 2.1: *VeriFlow* sits between the SDN applications and devices to intercept and check every rule entering the network.

2.2.1 Slicing the network into equivalence classes

One way to verify network properties is to prepare a model of the entire network using its current data-plane state, and run queries on this model [16, 47]. However, checking the entire network's state every time a new flow rule is inserted is wasteful, and fails to provide real-time response. Instead, we note that most forwarding rule changes affect only a small subset of all possible packets. For example, inserting a longest-prefix-match rule for the destination IP field will only affect forwarding for packets destined to that prefix. In order to confine our verification activities to only the affected set of packets, we slice the network into a set of equivalence classes (ECs) based on the new rule and the existing rules that overlap with the new rule. An equivalence class is defined as follows.

Definition (Equivalence Class): An equivalence class (EC) is a set P of packets such that for any $p_1, p_2 \in P$ and any network device R, the forwarding action is identical for p_1 and p_2 at R.

Separating the entire packet space into individual ECs allows *VeriFlow* to pinpoint the affected set of packets if a problem is discovered while verifying a newly inserted forwarding rule.

Let us look at an example. Consider an OpenFlow switch with two rules matching packets with destination IP address prefixes 11.1.0.0/16 and 12.1.0.0/16, respectively. If a new rule matching

destination IP address prefix 11.0.0.0/8 is added, it may affect packets belonging to the 11.1.0.0/16 range depending on the rules' priority values [11] (the longer prefix may not have higher priority). However, the new rule will not affect packets outside the range 11.0.0.0/8, such as 12.1.0.0/16. Therefore, *VeriFlow* will only consider the new rule (11.0.0.0/8) and the existing overlapping rule (11.1.0.0/16) while analyzing network properties. These two overlapping rules produce three ECs (represented using the lower and upper bound range values of the destination IP address field): 11.0.0.0 to 11.0.255.255, 11.1.0.0 to 11.1.255.255, and 11.2.255.255 to 11.255.255.255.

VeriFlow needs an efficient data structure to quickly store new network rules, find overlapping rules, and compute the affected ECs. For this we utilize a *multi-dimensional prefix tree (trie)* inspired by traditional packet classification algorithms [68].

A trie is an ordered tree data structure that stores an associative array. In our case, the trie associates the set of packets matched by a forwarding rule with the forwarding rule itself. Each level in the trie corresponds to a specific bit in a forwarding rule (equivalently, a bit in the packet header). Each node in our trie has three branches, corresponding to three possible values that the rule can match: 0, 1, and * (wildcard). The trie can be seen as a composition of several *sub-tries* or dimensions, each corresponding to a packet header field. We maintain a sub-trie in our multi-dimensional trie for each of the mandatory match and packet header fields supported by OpenFlow 1.1.0.[2] (Note that an optimization in our implementation uses a condensed set of fields in the trie; see § 2.3.2.) For example, the sub-trie representing the IPv4 destination corresponds to 32 levels in the trie. One of the sub-tries (DL_SRC in our design) appears at the top, the next field's sub-tries are attached to the leaves of the first, and so on (Figure 2.2). A path from the trie's root to a leaf of one of the bottommost sub-tries thus represents the set of packets that a rule matches. Each leaf stores the rules that match that set of packets, and the devices at which they are located (Figure 2.2).

When a new forwarding rule is generated by the application, we perform a lookup in our trie, by traversing it dimension by dimension to find all the rules that intersect the new rule. At each dimension, we narrow down the search area by only traversing those branches that fall within the range of the new rule using the field value of that particular dimension. The lookup procedure

[2](DL_SRC, DL_DST, NW_SRC, NW_DST, IN_PORT, DL_VLAN, DL_VLAN_PCP, DL_TYPE, NW_TOS, NW_PROTO, TP_SRC, TP_DST, MPLS_LABEL and MPLS_TC).

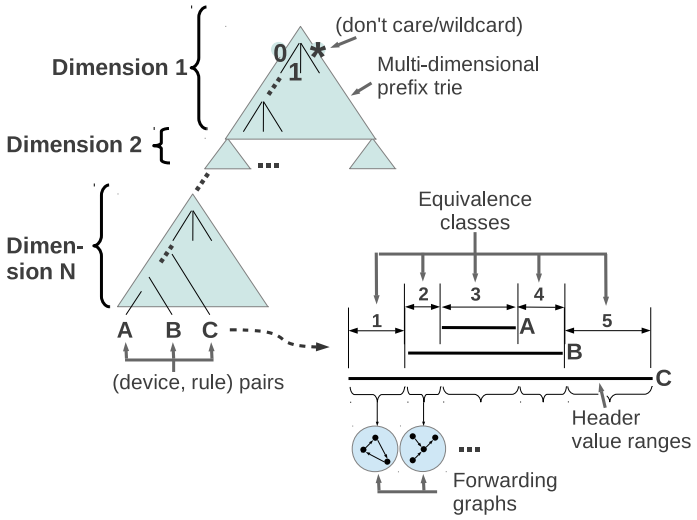


Figure 2.2: *VeriFlow*'s core algorithmic process.

results in the selection of a set of leaves of the bottommost dimension, each with a set of forwarding rules. These rules collectively define a set of packets (in particular, their corresponding forwarding rules) that could be affected by the incoming forwarding rule. This set may span multiple ECs. We next compute the individual ECs as illustrated in Figure 2.2. For each field, we find a set of disjoint ranges (lower and upper bound) such that no rule splits one of the ranges. An EC is then defined by a particular choice of one of the ranges for each of the fields. This is not necessarily a minimal set of ECs; for example, ECs 2 and 4 in Figure 2.2 could have been combined into a single EC. However, this method performs well in practice.

2.2.2 Modeling forwarding state with forwarding graphs

For each EC computed in the previous step, *VeriFlow* generates a *forwarding graph*. Each such graph is a representation of how packets within an EC will be forwarded through the network. In the graph, a node represents an EC at a particular network device, and a directed edge represents a forwarding decision for a particular (EC, device) pair. Specifically, an edge $X \rightarrow Y$ indicates that according to the forwarding table at node X, packets within this EC are forwarded to Y. To build the graph for each EC, we traverse our trie a second time to find the devices and rules that match packets from that EC. The second traversal is needed to find all those rules that were not necessary to compute the affected ECs in the first traversal, yet can still influence their forwarding behavior.

For example, for a new rule with 10.0.0.0/8 specified as the destination prefix, an existing 0.0.0.0/0 rule will not contribute to the generation of the affected ECs, but may influence their forwarding behavior depending on its priority. Given the range values of different fields of an EC, looking up matching rules from the trie structure can be performed very quickly. Here, *VeriFlow* only has to traverse those branches of the trie having rules that can match packets of that particular EC.

2.2.3 Running queries

Above, we described how *VeriFlow* models the behavior of the network using forwarding graphs, building forwarding graphs only for those equivalence classes (ECs) whose behavior may have changed. Next, we answer queries (check invariants) using this model.

VeriFlow maintains a list of invariants to be checked. When ECs have been modified, *VeriFlow* checks each (invariant, modified EC) pair. An invariant is specified as a verification function that takes as input the forwarding graph for a specific EC, performs arbitrary computation, and can trigger resulting actions. *VeriFlow* exposes an API (Application Programming Interface), the implementation of which is described in § 2.3.3, so that new invariants can be written and plugged in.

Up to a certain level of detail, the forwarding graph is an exact representation of the forwarding behavior of the network. Therefore, invariant modules can check a large diversity of conditions concerning network behavior. For example:

- Basic reachability: The verification function traverses the directed edges in the forwarding graph (using depth-first search in our implementation) to determine whether packets will be delivered to the destination address specified in the rule.

- Loop-freeness: The verification function traverses the given EC's forwarding graph to check that it does not contain a loop.

- Consistency: Given two (pre-specified) routers R_1, R_2 that are intended to have identical forwarding operations, the verification function traverses the forwarding graph starting at R_1 and R_2 to test whether the fate of packets is the same in both cases. (Any difference may indicate a bug.)

Further examples include detecting "black holes" where packets are dropped, ensuring isolation of multiple VLANs, verifying access control policies, checking whether a new rule conflicts with an existing rule, checking whether an EC changes its next hop due to the insertion/deletion of a rule, ensuring that packets always traverse a firewall, and so on.

There are two key limitations on what invariants can be feasibly implemented. First, *VeriFlow*'s forwarding graph construct must include the necessary information. Our current implementation of *VeriFlow* does not, for example, incorporate information on buffer sizes that would be necessary for certain performance invariants. (There is not, however, any fundamental reason that *VeriFlow* could not be augmented with such metadata.) Second, the invariant check must be implementable in the *incremental* manner described above where only the modified ECs are considered at each step.

If a verification function finds a violated invariant, it can choose to trigger further actions within *VeriFlow*. Two obvious actions are dropping the rule that was being inserted into the network, or installing the rule but generating an alarm for the operator. For example, the operator could choose to drop rules that cause a security violation (such as packets leaking onto a protected VLAN), but only generate an alarm for a black hole. Since verification functions are arbitrary code, they may take other actions as well, such as maintaining statistics (e.g., rate of forwarding behavior change) or writing logs.

2.2.4 Dealing with high verification time

VeriFlow achieves real-time response by confining its verification activities within those parts of the network that are affected when a new forwarding rule is installed. In general, the effectiveness of this approach will be determined by numerous factors, such as the complexity of verification functions, the size of the network, the number of rules in the network, the number of unique ECs covered by a new rule, the number of header fields used to match packets by a new rule, and so on.

However, perhaps the most important factor summarizing verification time is the *number of ECs modified*. As our later experiments will show, *VeriFlow*'s verification time is roughly linear in this number. In other words, *VeriFlow* has difficulty verifying invariants in real-time when large swaths of the network's forwarding behavior are altered in one operation.

When such disruptive events occur, *VeriFlow* may need to let new rules be installed in the network without waiting for verification, and run the verification process in parallel. We lose the ability to block problematic rules before they enter the network, but we note several mitigating facts. First, the most prominent example of a disruptive event affecting many ECs is a link failure, in which case *VeriFlow* anyway cannot block the modification from entering the network. Second, upon (eventually) detecting a problem, *VeriFlow* can still raise an alarm and remove the problematic rule(s) from the network. Third, the fact that the number of affected ECs is large may itself be worthy of an immediate alarm even before invariants are checked for each EC. Finally, our experiments with realistic forwarding rule update traces (§ 2.4) show that disruptive events (i.e., events affecting large number of ECs) are rare: in the vast majority of cases (around 99%), the number of affected ECs is small (less than 10).

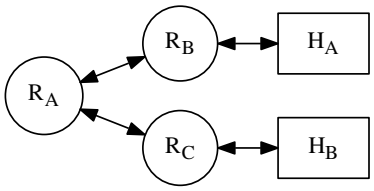
2.2.5 Alternative BDD-based design

To show the feasibility of this approach, we also studied a BDD-based network verification approach for comparison.

Representing OpenFlow rules as BDDs

The insight behind the BDD-based approach is that network forwarding behavior can be expressed as a boolean function, where variables represent the network's inputs (the location and packet contents injected into the network), the function's result represents the network's outputs (the location and packet contents sent out from the network), and the function's form represents packet forwarding behavior within the network (e.g., which packets are filtered/modified, and which paths they are sent on). As BDDs can be used to represent arbitrary boolean functions, to express the network as a BDD, we used similar techniques as used in [17]. We consider each bit of the packet header, which are used for rule matching, as individual variables in the BDD and build boolean functions using them to represent the forwarding behavior of a particular device. In order to limit the size of the model, once the set of affected rules of a network update are found, we generate a BDD corresponding to them.

Figure 2.3 shows an example of this process. In Figure 2.3a, we have a small network with



(a) Network topology

Router R_A	
Destination address	Next hop address
H_A	R_B
H_B	R_C

(b) Forwarding table at router R_A

(1) H_A & R_B
(2) H_B & R_C
(3) (H_A & R_B) | (H_B & R_C)
(4) ((H_A & R_B) | (H_B & R_C)) & R_A

(c) Boolean functions representing the forwarding table

Figure 2.3: *Representing forwarding table information as boolean functions.*

three routers and two hosts. Figure 2.3b shows the forwarding table (Forwarding Information Base or FIB) of router R_A. This FIB contains the forwarding information to reach hosts H_A and H_B. Boolean functions corresponding to the forwarding behavior of R_A are shown in Figure 2.3c. At R_A, if a packet arrived with destination address of H_A, it will be forwarded to R_B. Therefore, we AND the address representing H_A with the address of R_B (1). Similar action will be taken for packets destined to H_B (via R_C) (2). Now, we OR these two expressions as a packet can be destined to either of these two hosts (3). Here, for simplicity we assume that packets are sent only to these two hosts. Finally, these actions will be taken whenever a packet is at R_A. Therefore, we AND the address of R_A with the expression obtained in step 3 (4). Step 4 in Figure 2.3c represents the boolean function that represents the FIB state of a single router (R_A in this example). As a packet can be present at any one router at a particular time while it is present in the network, we OR together individual routers' FIB boolean functions to get the boolean function representing the entire network's state at a given moment.

Checking invariants

The next step is to check different network properties by running queries on the BDD built in the previous step. Given a particular state of the network and a set of new rules, we are interested to know whether inserting these rules will violate any network invariant or not. Therefore, we look for instances of packets that may fall into routing loops or may experience black holes. We express these queries as Computation Tree Logic (CTL) formulas as CTL provides a nice and compact way to explore states of a system represented as a finite state machine. We use standard CTL operator (EX, AX, AF, etc.) to express different network invariants that we are interested to check [17], and run the resulting query on the BDD structure built from network forwarding state. The results of these queries give us indications of potential violation of key network invariants and examples of packet instances that may cause an invariant to fail.

2.3 Implementation

We describe three key aspects of our implementation: our shim layer to intercept network events (§ 2.3.1), an optimization to accelerate verification (§ 2.3.2), and our API for custom invariants (§ 2.3.3). In addition to the prototype of *VeriFlow*, we also implemented a BDD-based design (§ 2.2.5) for comparison purpose.

2.3.1 Making deployment transparent

In order to ease the deployment of *VeriFlow* in networks with OpenFlow-enabled devices, and to use *VeriFlow* with unmodified OpenFlow applications, we need a mechanism to make *VeriFlow* transparent so that these existing OpenFlow entities may remain unaware of the presence of *VeriFlow*. We built two versions of *VeriFlow*. One is a *proxy process* [65] that sits between the controller and the network, and is therefore independent of the particular controller. The second version is *integrated* with the NOX OpenFlow controller [29] to improve performance; our performance evaluation is of this version. We expect one could similarly integrate *VeriFlow* with other controllers, such as Floodlight [5], Beacon [3] and Maestro [21], without significant trouble.

We built our implementation within NOX version 0.9.1 (full beta single-thread version). We

integrated *VeriFlow* within NOX, enabling it to run as a transparent rule verifier sitting between the OpenFlow applications implemented using NOX's API, and the switches and routers in the network. SDN applications running on NOX use the NOX API to manipulate the forwarding state of the network, resulting in OFPT_FLOW_MOD (flow table modification) and other OpenFlow messages generated by NOX. We modified NOX to intercept these messages, and redirect them to our *VeriFlow* module. This ensures that all messages are intercepted by *VeriFlow* before they are dispatched to the network. *VeriFlow* then processes and checks the forwarding rules contained in these messages for correctness, and can block problematic flow rules.

To integrate the *VeriFlow* module, we extend two parts of NOX. First, within the core of NOX, the *send_openflow_command()* interface is responsible for adding (relaying) flow rules from OpenFlow applications to the switches. At the lower layers of NOX, *handle_flow_removed()* handles events that remove rules from switches, due to rule timeouts or commands sent by applications. Our implementation intercepts all messages sent to these two function calls, and redirects them to *VeriFlow*. To reduce memory usage and improve running time, we pass these messages via shallow copy.

There are five types of flow table modification messages that can be generated by OpenFlow applications: OFPFC_ADD, OFPFC_MODIFY_STRICT, OFPFC_DELETE_STRICT, OFPFC_MODIFY and OFPFC_DELETE. These rules differ in terms of whether they add, modify or delete a rule from the flow table. The strict versions match all the fields bit by bit, whereas the non-strict versions allow wildcards. Our implementation handles all these message types appropriately.

2.3.2 Optimizing the verification process

We use an optimization technique that exploits the way certain match and packet header fields are handled in the OpenFlow 1.1.0 specification. 10 out of 14 fields in this specification do not support arbitrary wildcards.[3] One can only specify an exact value or the special *ANY* (wildcard) value in these fields. We do not use separate dimensions in our trie to represent these fields, because we do not need to find multiple overlapping ranges for them. Therefore, we only maintain

[3]IN_PORT, DL_VLAN, DL_VLAN_PCP, DL_TYPE, NW_TOS, NW_PROTO, TP_SRC, TP_DST, MPLS_LABEL and MPLS_TC.

the trie structure for the other four fields (DL_SRC, DL_DST, NW_SRC and NW_DST). Due to this change, we generate the set of affected equivalence classes (ECs) in three steps. First, we use the trie structure to look for network-wide overlapping rules, and find the set of affected packets determined by the four fields that are represented by the trie. Each individual packet set we get from this step is actually a set of ECs that can be distinguished by the other 10 fields. Second, for each of these packet sets, we extract all the rules that can match packets of that particular class from the location/device of the newly inserted rule. We linearly go through all these rules to find non-overlapping range values for the rest of the fields that are not maintained in the trie structure. Thus, each packet set found in the first step breaks into multiple finer packet sets spanning all the 14 mandatory OpenFlow match and packet header fields. Note that in this step we only consider the rules present at the device of the newly inserted rule. Therefore, in the final step, as we traverse the forwarding graphs, we may encounter finer rules at other devices that will generate new packet sets with finer granularity. We handle them by maintaining sets of excluded packets as described in the next paragraph.

Each forwarding graph that we generate using our trie structure represents the forwarding state of a group of packet sets that can be distinguished using the 10 fields that do not support arbitrary wildcards. Therefore, while traversing the forwarding graphs, we only work on those rules that overlap with the newly inserted rule on these 10 fields. As we move from node to node while traversing these graphs, we keep track of the ECs that have been served by finer rules and are no longer present in the primary packet set that was generated in the first place. For example, in a device, a subset of a packet set may be served by a finer rule having higher priority than a coarser rule that serves the rest of that packet set. We handle this by maintaining a set of excluded packets for each forwarding action. Therefore, whenever we reach a node that answers a query (e.g., found a loop or reached a destination), the primary packet set minus the set of excluded packets gives the set of packets that experiences the result of the query.

2.3.3 API to write general queries

We expose a set of functions that can be used to write general queries in C++. Below is a list of these functions along with the required parameters.

GetAffectedEquivalenceClasses: Given a new rule, this function computes the set of affected ECs, and returns them. It also returns a set of sub-tries from the last dimension of our trie structure. Each sub-trie holds the rules that can match packets belonging to one of the affected ECs. This information can be used to build the forwarding graphs of those ECs. This function takes the following parameters.

- Rule: A newly inserted rule.
- *Returns:* Affected ECs.
- *Returns:* Sub-tries representing the last dimension, and holding rules that can match packets of the affected ECs.

GetForwardingGraph: This function generates and returns the forwarding graph for a particular EC. It takes the following parameters.

- EquivalenceClass: An EC whose forwarding graph will be computed.
- TrieSet: Sub-tries representing the last dimension, and holding rules that match the EC supplied as the first argument.
- *Returns:* Corresponding forwarding graph.

ProcessCurrentHop: This function allows the user to traverse a forwarding graph in a custom manner. Given a location and EC, it returns the corresponding next hop. It handles the generation of multiple finer packet sets by computing excluded packet sets that need to be maintained because of our optimization strategy (§ 2.3.2). Due to this optimization, this function returns a set of (next hop, excluded packet set) tuples — effectively, an annotated directed edge in the forwarding graph. With repeated calls to this function across nodes in the forwarding graphs, custom invariant-checking modules can traverse the forwarding graph and perform arbitrary computation on its structure. This function takes the following parameters.

- ForwardingGraph: The forwarding graph of an EC.
- Location: The current location of the EC.
- *Returns:* (Next hop, excluded packet set) tuples.

Let us look at an example that shows how this API can be used in practice. A network operator may want to ensure that packets belonging to a certain set always pass through a firewall device. This invariant can be violated during addition/deletion of rules, or during link up/down events. To

check this invariant, the network operator can extend *VeriFlow* using the above API to incorporate a custom query algorithm that generates an alarm when the packet set under scrutiny bypasses the firewall device. In fact, the network operator can implement any query that can be answered using the information present in the forwarding graphs.

2.4 Evaluation

In this section, we present a performance evaluation of our *VeriFlow* implementation. As *VeriFlow* intercepts every rule insertion message whenever it is issued by an SDN controller, it is crucial to complete the verification process in real time so that network performance is not affected, and to ensure scalability of the controller. We evaluated the overhead of *VeriFlow*'s operations with the help of two experiments. In the first experiment (§ 2.4.1), our goal is to microbenchmark different phases of *VeriFlow*'s operations and observe their contribution to the overall running time. The goal of the second experiment (§ 2.4.2) is to assess the impact of *VeriFlow* on TCP connection setup latency and throughput as perceived by end users of an SDN.

In all of our experiments, we used our basic reachability algorithms to test for loops and black holes for every flow modification message that was sent to the network. All of our experiments were performed on a Dell Optiplex 9010 machine with an Intel Core i7 3770 CPU with 4 physical cores and 8 threads at 3.4 GHz, and 32 GB of RAM, running 64 bit Ubuntu Linux 11.10.

2.4.1 Per-update processing time

In this experiment, we simulated a network consisting of 172 routers following a Rocketfuel [12] topology (AS 1755), and replayed BGP (Border Gateway Protocol) RIB (Routing Information Base) and update traces collected from the Route Views Project [15]. We built an OSPF (Open Shortest Path First) simulator to compute the IGP (Interior Gateway Protocol) path cost between every pair of routers in the network. A BGP RIB snapshot consisting of 5 million entries was used to initialize the routers' FIB (Forwarding Information Base) tables. Only the FIBs of the border routers were initialized in this phase. We randomly mapped Route Views peers to border routers in our network, and then replayed RIB and update traces so that they originate according to this mapping. We replayed a BGP update trace containing 90,000 updates to trigger dynamic

changes in the network. Upon receiving an update from the neighboring AS, each border router sends the update to all the other routers in the network. Using standard BGP polices, each router updates its RIB using the information present in the update, and updates its FIB based on BGP AS path length and IGP path cost. We fed all the FIB changes into *VeriFlow* to measure the time *VeriFlow* takes to complete its individual steps described in § 2.2. We recorded the run time to process each change individually. Note that in this first set of experiments, only the destination IP address is used to forward packets. Therefore, only this one field contributes to the generation of equivalence classes (ECs). We initialize the other fields with ANY (wildcards).

The results from this experiment are shown in Figure 2.4(a). *VeriFlow* is able to verify most of the updates within 1 millisecond (ms), with mean verification time of 0.38ms. Moreover, of this time, the query phase takes only 0.01ms on an average, demonstrating the value of reducing the query problem to a simple graph traversal for each EC. Therefore, *VeriFlow* would be able to run multiple queries of interest to the network operator (e.g., black hole detection, isolation of multiple VLANs, etc.) within a millisecond time budget.

We found that the number of ECs that are affected by a new rule strongly influences verification time. The scatter plot of Figure 2.4(b) shows one data point for each observed number of modified ECs (showing the mean verification time across all rules, which modified that number of ECs). The largest number of ECs affected by a single rule was 574; the largest verification latency was 159.2ms due to an update affecting 511 ECs. However, in this experiment, we found that for most updates the number of affected ECs is small. 94.5% of the updates only affected a single EC, and 99.1% affected less than 10 ECs. Therefore, only a small fraction of rules (0.9%) affected large numbers of ECs. This can be observed by looking at the long tail of Figure 2.4(a).

In the above experiment, we assumed that the network topology remains unchanged, i.e., there are no link or node failures. In case of a link failure or node failure (which can be thought of as failure of multiple links connected to the failed node), the packets that were using that link or node will experience changes in their forwarding behavior. When this happens, *VeriFlow*'s job is to verify the fate of those affected packets. In order to evaluate *VeriFlow*'s performance in this scenario, we used the above topology and traces to run a new experiment. In this experiment, we fed both the BGP RIB trace and update trace to the network. Then we removed each of the

packet-carrying links (381 in total) of the network one by one (restoring a removed link before removing the next), and computed the number of affected ECs and the running time of *VeriFlow* to verify the behavior of those classes. We found that most of the link removals affected a large number of ECs. 254 out of 381 links affected more that 1,000 ECs. The mean verification time to verify a link failure event was 1.15 seconds, with a maximum of 4.05 seconds. We can deal with such cases by processing the forwarding graphs of different ECs in parallel on multi-core processors. This is possible because the forwarding graphs do not depend on each other, or on any shared data structure. However, as link or node failures cannot be avoided once they happen, this may not be a serious issue for network operators.

In order to evaluate *VeriFlow*'s performance in the presence of more fields, we changed the input data set to add packet filters that will selectively drop packets after matching them against multiple fields. We randomly selected a subset of the existing RIB rules currently present in the network, and inserted packet filter rules by specifying values in some of the other fields that were not present in the original trace. We ran this experiment with two sets of fields. In the first set we used TP_SRC and TP_DST in addition to NW_DST (3 fields in total), which was already present in the trace. For each randomly selected RIB rule, we set random values to those two fields (TP_SRC and TP_DST), and set its priority higher than the original rule. The remaining 11 fields are set to ANY. While replaying the updates, all the 14 fields except NW_DST are set to ANY.

In the second set we used NW_SRC, IN_PORT, DL_VLAN, TP_SRC and TP_DST in addition to NW_DST (6 fields in total). For each randomly selected RIB rule, we set random values to IN_PORT, DL_VLAN, TP_SRC and TP_DST, a random /16 value in NW_SRC, and set the priority higher than the original rule. The remaining 8 fields are set to ANY. While replaying the updates, all the 14 fields except NW_SRC and NW_DST are set to ANY. In the updates, the NW_SRC is set to a random /12 value and the NW_DST is the original value present in the trace. We ran this experiment multiple times varying the percentage of RIB rules that are used to generate random filter rules with higher priority.

Figure 2.4(c) shows the results of this experiment. Verification time is heavily affected by the number of fields used to classify packets. This happens because as we use more fields to classify packets at finer granularities, more unique ECs are generated, and hence more forwarding graphs

need to be verified. We also note from Figure 2.4(c) that *VeriFlow*'s overall performance is not affected much by the number of filters that we install into the network.

In all our experiments thus far, we kept a fixed order of packet header fields in our trie structure. We started with DL_SRC (DS), followed by DL_DST (DD), NW_SRC (NS) and NW_DST (ND). In order to evaluate the performance of *VeriFlow* with different field orderings, we re-ran the above packet filter experiment with reordered fields. In all the runs we used random values for the NW_SRC field and used the NW_DST values present in the Route Views traces. All the other fields were set to ANY. We installed random packet filter rules for 10% of the BGP RIB entries. As our dataset only specified values for the NW_SRC and NW_DST fields, there were a total of 12 different orderings of the aforementioned 4 fields. Table 2.1 shows the results from this experiment.

Table 2.1: Effect of different field orderings on total running time of *VeriFlow*.

Order	Time (ms)	Order	Time (ms)
DS-DD-NS-ND	1.001	DS-DD-ND-NS	0.090
DS-NS-DD-ND	1.057	DS-ND-DD-NS	0.096
NS-DS-DD-ND	1.144	ND-DS-DD-NS	0.101
NS-DS-ND-DD	1.213	ND-DS-NS-DD	0.103
NS-ND-DS-DD	1.254	ND-NS-DS-DD	0.15
DS-NS-ND-DD	1.116	DS-ND-NS-DD	0.098

From Table 2.1, we can see that changing the field order in the trie structure greatly influences the running time of *VeriFlow*. Putting the NW_DST field ahead of NW_SRC reduced the running time by an order of magnitude (from around 1ms to around 0.1ms). This happens because a particular field order may produce fewer unique ECs compared to other field orderings for the same rule. However, it is difficult to come up with a single field order that works best in all scenarios, because it is highly dependent on the type of rules present in a particular network. Changing the field order in the trie structure dynamically and efficiently as the network state evolves would be an interesting area for future work.

Checking non-reachability invariants: Most of our discussion thus far focused on checking invariants associated with the inter-reachability of network devices. To evaluate the generality of our tool, we implemented two more invariants using our API that were not directly related to reachability: *conflict detection* (whether the newly inserted rule violates isolation of flow tables

between network slices, accomplished by checking the output of the EC search phase), and *k-monitoring* (ensuring that all paths in the network traverse one of several deployed monitoring points, done by augmenting the forwarding graph traversal process). We found that the overhead of these checks was minimal. For the conflict detection query, we ran the above filtering experiment using the 6-field set with 10% and 20% newly inserted random rules. However, this time instead of checking the reachability of the affected ECs as each update is replayed, we only computed the set of rules that overlap/conflict with the newly inserted rule. The results from this experiment are shown in Figure 2.4(d).

From this figure, we can see that conflicting rule checking can be done quickly, taking only 0.305ms on average. (The step in the CDF is due to the fact that some withdrawal rules did not overlap with any existing rule.)

For the k-monitoring query experiment, we used a snapshot of the Stanford backbone network data-plane state that was used in [40]. This network consists of 16 routers, where 14 of these are internal routers and the other 2 are gateway routers used to access the outside network. The snapshot contains 7,213 FIB table entries in total. In this experiment, we used $VeriFlow$ to test whether *all* the ECs currently present in the network pass through one of the two gateway routers of the network. We observed that at each location the average latency to perform this check for all the ECs is around 68.06ms with a maximum of 75.39ms.

2.4.2 Effect on network performance

In order to evaluate the effect of $VeriFlow$'s operations on user-perceived TCP connection setup latency and the network throughput, we emulated an OpenFlow network consisting of 172 switches following the aforementioned Rocketfuel topology using Mininet [7]. Mininet creates a software-defined network (SDN) with multiple nodes on a single machine. We connected one host to every switch in this emulated network. We ran the NOX OpenFlow controller along with an application that provides the functionality of a learning switch. It allows a host to reach any other host in the network by installing flow rules in the switches using flow modification (Flow_Mod) messages. We implemented a simple TCP server program and a simple TCP client program to drive the experiment. The server program accepts TCP connections from clients and closes the connection

immediately. The client program consists of two threads. The primary thread continuously sends connect requests to a random server using a non-blocking socket. To vary the intensity of the workload, our TCP client program generates connections periodically with a parameterized sleep interval (S). The primary thread at each client sleeps for a random interval between 0 to S seconds (at microsecond granularity) before initiating the connection request, and iterating. The secondary thread at each client uses the *select* function to look for connections that are ready for transmission or experienced an error. A user supplied polling interval (P) is used to control the rate at which the select call will return. We set P inversely proportional to the S value to avoid busy waiting and to allow the other processes (e.g., Open vSwitch [10]) to get a good share of the CPU. We ran the server program at each of the 172 hosts, and configured the client programs at all the hosts to continually connect to the server of random hosts (excluding itself) over a particular duration (at least 10 minutes). In the switch application, we set the rule eviction idle timeout to 1 second and hard timeout to 5 seconds.

We ran this experiment first with NOX alone, and then with NOX and $VeriFlow$. We used the same seed in all the random number generators to ensure similar loads in both the runs. We also varied the S value to monitor the performance of $VeriFlow$ under a range of network loads.

Figure 2.5(a) shows the number of TCP connections that were successfully completed per second for different workloads both with and without $VeriFlow$. From this figure, we can see that in all the cases $VeriFlow$ imposes negligible overhead on the TCP connection setup throughput in our emulated OpenFlow network. The largest reduction in throughput that we observed in our experiments was only 0.74%.

Figure 2.5(b) shows the number of flow modification (Flow_Mod) messages that were processed and sent to the network per second for different workloads both with and without $VeriFlow$. From this figure, again we can see that in all the cases $VeriFlow$ imposes minimal overhead on the flow modification message throughput. The largest reduction in throughput that we observed in our experiments was only 12.8%. This reduction in throughput is caused by the additional processing time required to verify the flow modification messages before they are sent to the network.

In order to assess the impact of $VeriFlow$ on end-to-end TCP connection setup latency, we ran this experiment with S set to 30 seconds. We found that in the presence of $VeriFlow$, the

average TCP connection setup latency increases by 15.5% (45.58ms without *VeriFlow* versus 52.63ms with *VeriFlow*). As setting up a TCP connection between two hosts in our emulated 172 host OpenFlow network requires installing flow rules into more than one switch, the verification performed by *VeriFlow* after receiving each flow rule from the controller inflates the end-to-end connection setup latency to some extent.

Lastly, we ran this experiment after modifying *VeriFlow* to work with different numbers of OpenFlow packet header fields. Clearly, if we restrict the number of fields during the verification process, there will be less work for *VeriFlow*, resulting in faster verification time. In this experiment, we gradually increased the number of OpenFlow packet header fields that were used during the verification process (from 1 to 14). *VeriFlow* simply ignored the excluded fields, and it reduced the number of dimensions in our trie structure. We set S to 10 seconds and ran each run for 10 minutes. During the runs, we measured the verification latency experienced by each flow modification message generated by NOX, and computed their average at each run.

Figure 2.5(c) shows the results from this experiment. Here, we see that with the increase in the number of packet header fields, the verification overhead of *VeriFlow* increases gradually but always remains low enough to ensure real-time response. The 5 fields that contributed most in the verification overhead are DL_SRC, DL_DST, NW_SRC, NW_DST and DL_TYPE. This happened because these 5 fields had different values at different flow rules, and contributed most in the generation of multiple ECs. The other fields were mostly wildcards, and did not generate additional ECs.

2.4.3 Comparison with related work

Finally, we compared performance of our technique with two pieces of related work: the Hassel tool presented in [40] (provided to us by the authors), and a BDD-based analysis tool (as described in § 2.2.5) that we implemented from scratch following the strategy presented in [17] (the original code was not available to us). The authors of [40] provided two copies of their tool, one in Python and one in C, and we evaluated using the better-performing C version. While we note these works solve different problems from our work (e.g., HSA performs static verification, and does it between port pairs), we present these results to put *VeriFlow*'s performance in context. First, we ran

Hassel over the snapshot of the Stanford backbone network data-plane state that was used in [40]. We found that Hassel's average time to check reachability between a pair of ports (effectively exploring all ECs for that source-destination pair) was 578.62ms, with a maximum of 6.24 seconds. In comparison, *VeriFlow* took only 68.06ms on average (with a maximum of 75.39ms) to test the reachability of all the ECs currently present at a single node in the network. Next, in the BDD-based approach, we used the NuSMV [9] model checker to build a BDD using a new rule and the overlapping existing rules, and used CTL (Computation Tree Logic) to run reachability queries [17]. Here, we used the Rocketfuel topology and Route Views traces that we used in our earlier experiments. We found that this approach is quite slow and does not provide real-time response while inserting and checking new forwarding rules. Checking an update took 335.71ms on an average with a maximum of 67.16 seconds.

2.5 Summary

In this chapter, we presented *VeriFlow*, a network debugging tool to find faulty rules issued by SDN applications, and optionally prevent them from reaching the network and causing anomalous network behavior. *VeriFlow* leverages a set of efficient algorithms to check rule modification events in real time before they are sent to the live network. To the best of our knowledge, *VeriFlow* is the first tool that can verify network-wide invariants in a live network in real time. With the help of experiments using a real world network topology, real world traces, and an emulated OpenFlow network, we found that *VeriFlow* is capable of processing forwarding table updates in real time.

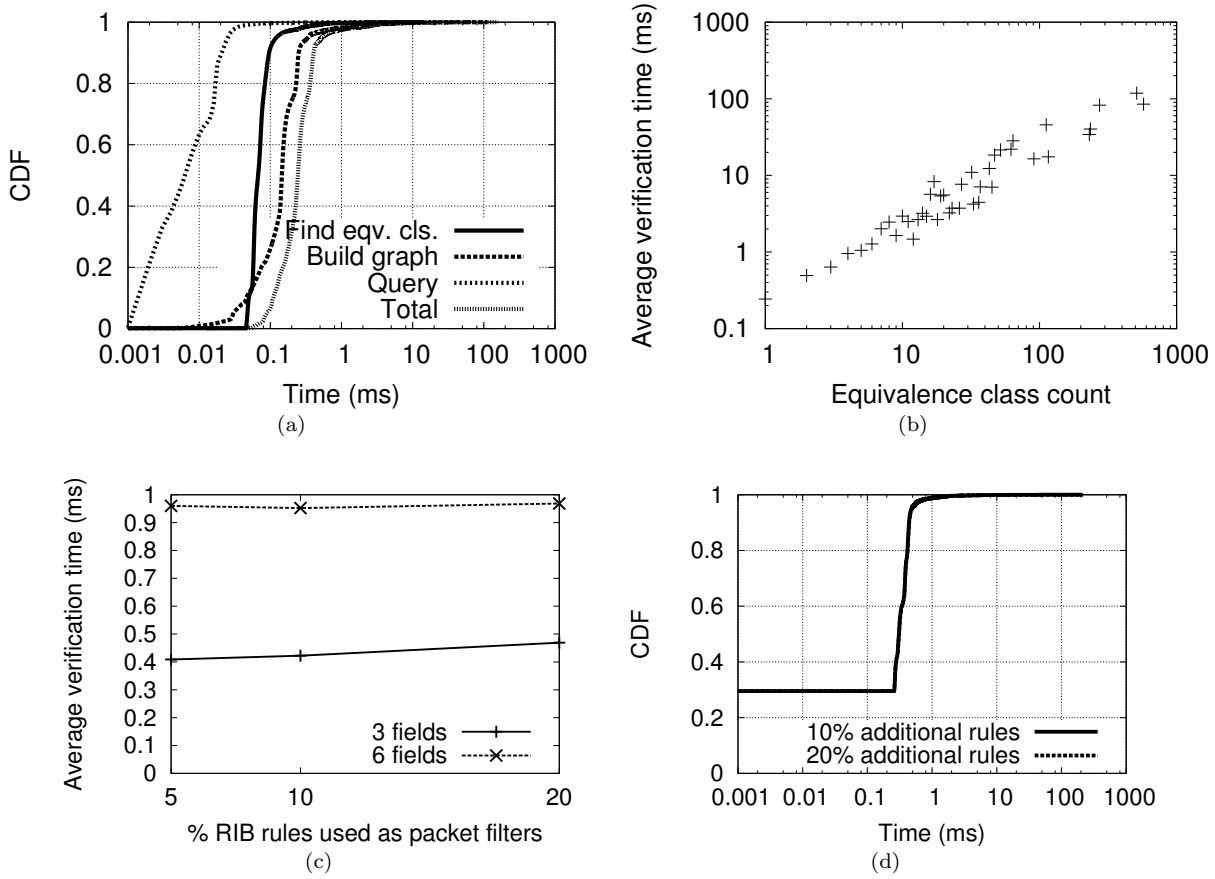


Figure 2.4: Per-update processing times: (a) Microbenchmark results, using the Route Views trace. Total verification time of *VeriFlow* remained below 1ms for 97.8% of the updates. (b) Scatter plot showing the influence of number of equivalence classes on verification time. (c) Results from multi-field packet filter experiment using the Route Views trace. As more fields are used in forwarding rules, the running time of *VeriFlow* increases. The average verification latency is not significantly influenced as we increase the number of filters present in the network. (d) Results from the conflict detection test. *VeriFlow* is fast enough to compute all the conflicting rules within hundreds of microseconds for 99% of the updates.

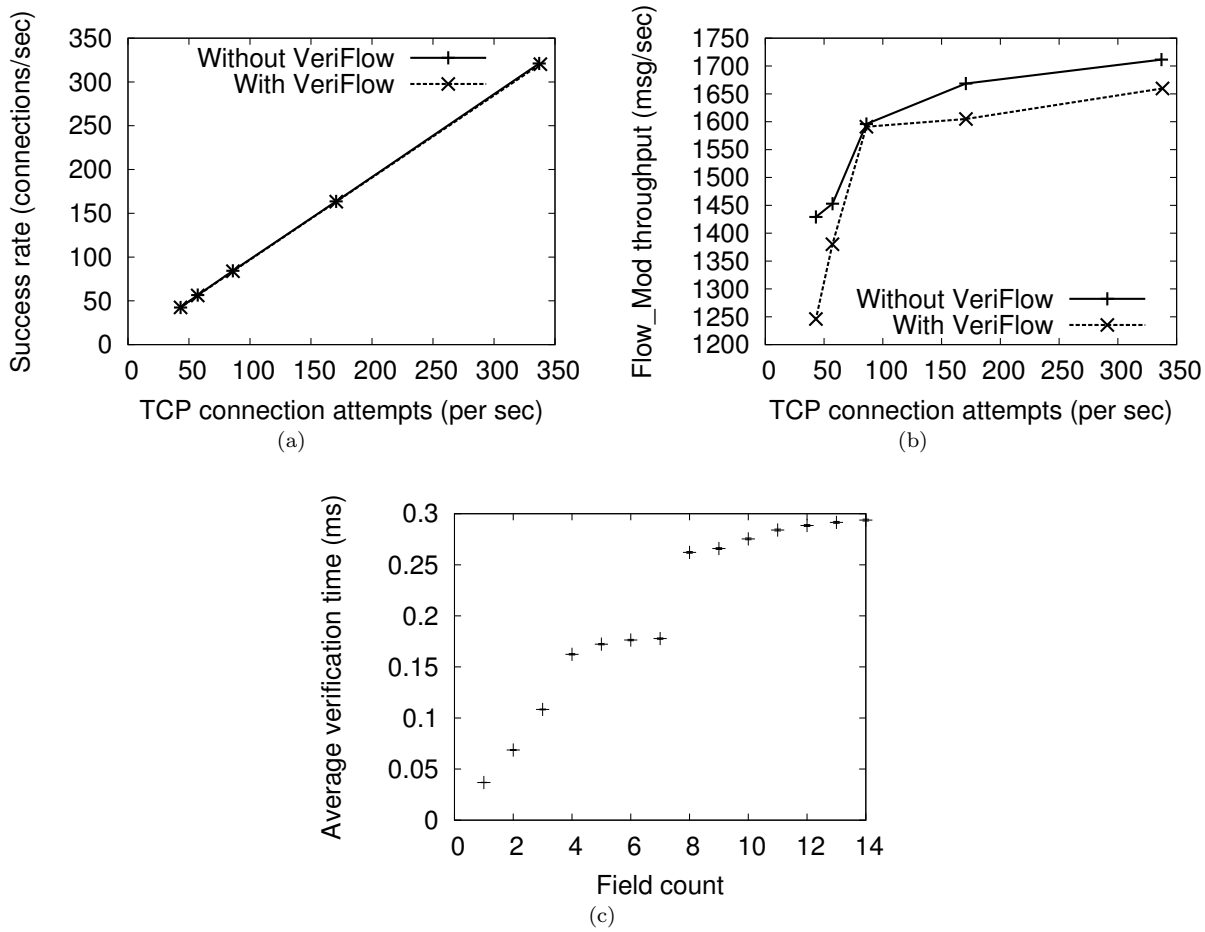


Figure 2.5: Effect on network performance: (a) TCP connection setup throughput, and (b) Throughput of flow modification (Flow_Mod) messages, with and without $VeriFlow$. For different loads, $VeriFlow$ imposes minimal overhead. (c) Effect of the number of packet header fields on $VeriFlow$'s verification speed. As we increase the number of fields, overhead of $VeriFlow$ increases gradually.

Chapter 3

Synthesizing Update Timing and Ordering

Network operators often establish a set of correctness conditions to ensure successful operation of the network, such as the preference of one path over another, the prevention of untrusted traffic from entering a secure zone, or loop and black hole avoidance. As networks become an increasingly crucial backbone for critical services, the ability to construct networks that obey correctness criteria is becoming even more important. Moreover, as modern networks are continually changing, it is critical for them to be correct even during transitions. Thus, a key challenge is to guarantee that properties are preserved during transitions from one correct configuration to a new correct configuration, which has been referred as *network consistency* [63].

Several recent proposed systems [34, 38, 45, 63] consistently update software-defined networks (SDNs), transitioning between two operator-specified network snapshots. However, those methods maintain only *specific* properties, and can substantially delay the network update process. Consistent updates [63] (CU), for example, only guarantees *coherence*: during a network update any packet or any flow is processed by either a new or an old configuration, but never by a mix of the two. This is a relatively strong policy that is sufficient to guarantee a large class of more specific policies (no loop, firewall traversal, etc.), but it comes at the cost of requiring a two-phase update mechanism that incurs substantial delay between the two phases and doubles flow entries temporarily. For networks that care only about a weaker consistency property, e.g., only loop freedom, this overhead is unnecessary. At the same time, networks sometimes need properties *beyond* what CU provides: CU only enforces properties on individual flows, but not across flows (e.g., "no more than two flows on a particular link"). SWAN [34] and zUpdate [45] also ensure only a specific property, in their case congestion freedom.

That leads to a question: is it possible to efficiently maintain *customizable* correctness policies as the network evolves? Ideally, we want the "best of both worlds": the efficiency of simply

immediately installing updates without delay, but the safety of whatever correctness properties are relevant to the network at hand.

We are not the first to define this goal. Recently, Dionysus [36] proposed to reduce network update time to just what is necessary to satisfy a certain property. However, Dionysus requires a rule dependency graph for each particular invariant, produced by an algorithm specific to that invariant (the paper presents an algorithm for packet coherence). For example, a waypointing invariant would need a new algorithm. Furthermore, the algorithms work only when forwarding rules match exactly one flow.

We take a different approach that begins with an observation: synthesizing consistent updates for arbitrary consistency policies is hard, but network verification on general policies is comparatively easy, especially now that real-time data plane verification tools [16, 39, 43] can verify very generic data-plane properties of a network state within milliseconds. In fact, as also occurs in domains outside of networking, there is a connection between synthesis and verification. A feasible update sequence is one which the relevant properties are verifiable at each moment in time. Might a verifier serve as a guide through the search space of possible update sequences?

Based on that insight, we propose the design of one synthesis component in our system, the Customizable Consistency Generator (*CCG*), which efficiently and consistently updates SDNs under customizable properties (invariants), intuitively by converting the scheduling synthesis problem to a series of network verification problems. With *CCG*, network programmers can express desired invariants using an interface (from [43]) which allows invariants to be defined as essentially arbitrary functions of a data plane snapshot, generally requiring only a few tens of lines of code to inspect the network model. Next, *CCG* runs a greedy algorithm: when a new rule arrives from the SDN controller, *CCG* checks whether the network would satisfy the desired invariants if the rule were applied. If so, the rule is sent without delay; otherwise, it is buffered, and at each step *CCG* checks its buffer to see if any rules can be installed safely (via repeated verifications).

This simplistic algorithm has two key problems. First, the greedy algorithm may not find the best (e.g., fastest) update installation sequence, and even worse, it may get stuck with no update being installable without violating an invariant. However, we identify a fairly large scope of policies that are "segment-independent" for which the heuristic is guaranteed to succeed without deadlock

(§3.4.2). For non-segment-independent policies, *CCG* needs a more heavyweight update technique, such as Consistent Updates [63] or SWAN [34], to act as a fallback. But *CCG* triggers this fallback mechanism only when the greedy heuristic determines it cannot offer a feasible update sequence. This is very rare in practice for the invariants we test (§3.6), and even when the fallback is triggered, only a small part of the transition is left to be handled by it, so the overhead associated with the heavyweight mechanism (e.g., delay and temporarily doubled FIB entries) is avoided as much as possible.

The second challenge lies in the verifier. Existing real-time data plane verifiers, such as VeriFlow and NetPlumber, assume that they have an accurate network-wide snapshot; but the network is a distributed system and we cannot know exactly when updates are applied. To address that, *CCG* explicitly models the uncertainty about network state that arises due to timing, through the use of *uncertain forwarding graph* (§3.3), a data structure that compactly represents the range of possible network behaviors given the available information. Although compact, *CCG*'s verification engine produces potentially larger models than those of existing tools due to this "uncertainty" awareness. Moreover, as a subroutine of the scheduling procedure, the verification function is called much more frequently than when it is used purely for verification. For these reasons, a substantial amount of work went into optimization, as shown in §3.6.1.

In summary, our contributions are:

- We developed a system, *CCG*, to efficiently synthesize network update orderings to preserve customizable policies as network states evolve.
- We created a graph-based model to capture network uncertainty, upon which real-time verification is performed (90% of updates verified within 10 μs).
- We evaluate the performance of our *CCG* implementation in both emulation and a physical testbed, and demonstrate that *CCG* offers significant performance improvement over previous work—up to 3× faster updates, typically with zero extra FIB entries—while preserving various levels of consistency.

3.1 Problem Definition

We design *CCG* to achieve the following three objectives:

1) Consistency at Every Step. Network changes can occur frequently, triggered by the control applications, changes in traffic load, system upgrades, or even failures. Even in SDNs with a logically centralized controller, the asynchronous and distributed nature implies that no single component can always obtain a fully up-to-date view of the entire system. Moreover, data packets from all possible sources may traverse the network at any time in any order, interleaving with the network data plane updates. How can we continuously enforce consistency properties, given the incomplete and uncertain network view at the controller?

2) Customizable Consistency Properties. The range of desired consistency properties of networks is quite diverse. For example, the successful operations of some networks may depend on a set of paths traversing a firewall, certain "classified" hosts being unreachable from external domains, enforcement of access control to protect critical assets, balanced load across links, loop freedom, etc. As argued in [46], a generic framework to handle general properties is needed. Researchers have attempted to ensure certain types of consistency properties, e.g., loop freedom or absence of packet loss [34, 45], but those studies do not provide a generalized solution. Dionysus [36], as stated earlier, generalizes the scope of consistency properties it deals with, but still requires designing specific algorithms for different invariants. Consistent Updates [63] is probably the closest solution to support general consistency properties because it provides the relatively strong property of packet coherence which is sufficient to guarantee many other properties; but as we will see next, it sacrifices efficiency.

3) Efficient Update Installation. The network controller should react in a timely fashion to network changes to minimize the duration of performance drops and network errors. There have been proposals [34, 38, 45, 54, 63] that instill correctness according to a specific consistency property, but these approaches suffer substantial performance penalties. For example, the waiting time between phases using the two-phase update scheme proposed in CU [63] is at least the maximum delay across all the devices, assuming a completely parallel implementation. Dionysus [36] was recently proposed to update networks via dynamic scheduling atop a consistency-preserving dependency graph. However, it requires implementing a new algorithm and dependency graph for each new invariant to achieve good performance. For example, a packet coherence invariant needs one algorithm and a waypoint invariant would need another algorithm. In contrast, our

approach reduces the consistency problem to a general network verification problem, which can take a broad range of invariants as inputs. In particular, one only needs to specify the verification function instead of designing a new algorithm. This approach also grants *CCG* the ability to deal with wildcard rules efficiently, in the same way as general verification tools, whereas Dionysus only works for applications with exact match on flows or classes of flows.

3.2 Overview

CCG converts the update scheduling problem into a network verification problem. Our overall approach is shown in Figure 4.1. Our uncertainty-aware network model (§3.3.2) provides a compact symbolic representation of the different possible states the network could be in, providing input for the verification engine. The verification engine is responsible for verifying application updates against specified invariants and policies (§3.3.4). Based on verification results, *CCG* synthesizes an efficient update plan to preserve policy consistency during network updates, using the basic heuristic and a more heavyweight fallback mechanism as backup (§3.4.1 and §3.4.3). One key feature of *CCG* is that it operates in a *black-box* fashion, providing a general platform with a very flexible notion of consistency. For example, one can "plug in" a different verification function and a fallback update scheduling tool to meet one's customized needs.

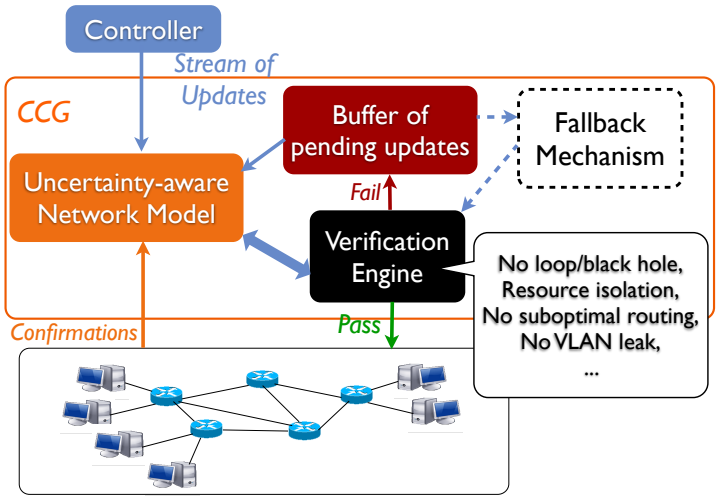


Figure 3.1: *System architecture of CCG.*

3.3 Verification under Uncertainty

We start by describing the problem of network uncertainty (§3.3.1), and then present our solution to model a network in the presence of uncertainty (§3.3.2 and §3.3.3). Our design centers around the idea of *uncertain forwarding graphs*, which compactly represent the entire set of possible network states from the standpoint of packets. Next, we describe how we use our model to perform uncertainty-aware network verification (§3.3.4).

3.3.1 The Network Uncertainty Problem

Networks must disseminate state among distributed and asynchronous devices, which leads to the inherent *uncertainty* that an observation point has in knowing the current state of the network. We refer to the time period during which the view of the network from an observation point (e.g., an SDN controller) might be inconsistent with the actual network state as *temporal network uncertainty*. The uncertainty could cause network behaviors to deviate from the desired invariants temporarily or even permanently.

Figure 3.2 shows a motivating example. Initially, switch A has a forwarding rule directing traffic to switch B. Now the operator wants to reverse the traffic by issuing two instructions in sequence: (1) remove the rule on A, and (2) insert a new rule (directing traffic to A) on B. But it is possible that the second operation finishes earlier than the first one, causing a transient loop that leads to packet losses. That is not an uncommon situation; for example, three out of eleven bugs found by NICE [22] (BUG V, IX and XI) are caused by the control programs' lack of knowledge of the network states.

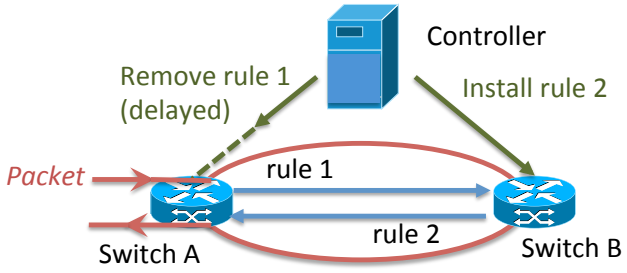


Figure 3.2: *Example: challenge of modeling networks in the presence of uncertainty.*

Such errors may have serious consequences. In the previous example, the resulting packet losses

could cause a significant performance drop. A recent study [**?**] shows TCP transfers with loss may take five times longer to complete. Other transient errors could violate security policy, e.g., malicious packets could enter a secure zone because of a temporary access control violation [63].

To make matters worse, errors caused by unawareness of network temporal uncertainty can be permanent. For instance, a control program initially instructs a switch to install one rule, and later removes that rule. The two instructions can be reordered at the switch [30], which ultimately causes the switch to install a rule that ought to be removed. The view of the controller and the network state will remain inconsistent until the rule expires. One may argue that inserting a barrier message in between the two instructions would solve the problem. However, this may harm performance because of increasing control traffic and switch operations. There are also scenarios in which carefully crafting an ordering does not help [63]. In addition, it is difficult for a controller to figure out when to insert the barrier messages. *CCG* addresses that by serializing only updates that have potential to cause race conditions that violate an invariant (§3.5).

3.3.2 Uncertainty Model

We first briefly introduce our prior work VeriFlow, a real-time network-wide data plane verifier. VeriFlow intercepts every update issued by the controller before it hits the network and verifies its effect in real time. VeriFlow first slices the set of possible packets into Equivalence Classes (ECs) of packets using all existing forwarding rules and the new update. Each EC is a set of packets that experiences the same forwarding actions throughout the network. Next, VeriFlow builds a *forwarding graph* for each EC affected by the update, by collecting forwarding rules influencing the EC. Lastly, VeriFlow traverses each of these graphs to verify network-wide invariants.

Naively, to model network uncertainty, for every update, we need two graphs to symbolically represent the network behavior with and without the effect of the update for each influenced EC, until the controller is certain about the status of the update. If n updates are concurrently "in flight" from the controller to the network, we would need 2^n graphs to represent all possible sequences of update arrivals. Such a state-space explosion will result in a huge memory requirement and excessive processing time to determine consistent update orderings.

To address that, we efficiently model the network forwarding behavior as a *uncertain forwarding*

graph, whose links can be marked as *certain* or *uncertain*. A forwarding link is *uncertain* if the controller does not yet have information on whether that corresponding update has been applied to the network. The graph is maintained by the controller over time. When an update is sent, its effect is applied to the graph and marked as uncertain. After receipt of an acknowledgment from the network that an update has been applied (or after a suitable timeout), the state of the related forwarding link is changed to *certain*. Such a forwarding graph represents all possible combinations of forwarding decisions at all the devices.

In this way, the extra storage required for uncertainty modeling is linearly bounded by the number of uncertain rules. We next examine when we can resolve uncertainty, either confirming a link as certain or removing it.

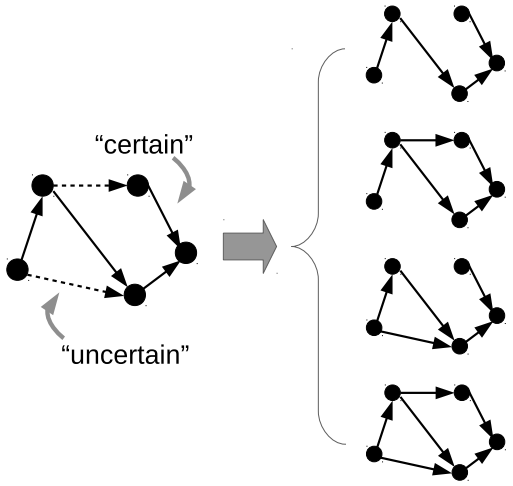


Figure 3.3: *CCG's uncertain forwarding graph.*

3.3.3 Dynamic Updating of the Model

In order to model the most up-to-date network state, we need to update the model as changes happen in the network. At first glance, one might think that could be done simply by marking forwarding links as uncertain when new updates are sent, and then, when an ack is received from the network, marking them as certain. The problem with that approach is that it may result in inconsistencies from the data packets' perspective. Consider a network consisting of four switches, as in Figure 3.4.

The policy to enforce is that packets from a particular source entering Switch s_1 should not

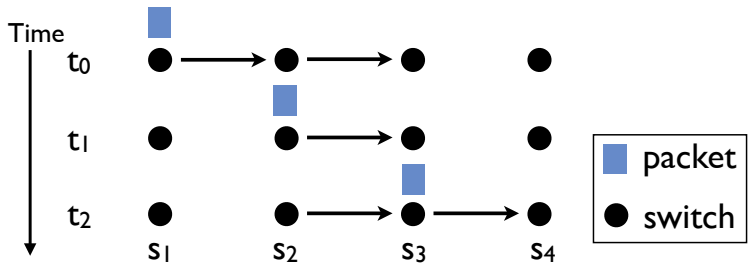


Figure 3.4: *Example: challenge of dealing with non-atomicity of packet traversal.*

reach Switch s_4. Initially, at time t_0, Switch s_3 has a filtering rule to drop packets from that source, whereas all the other switches simply pass packets through. The operator later wants to drop packets on s_1 instead of s_3. To perform the transition in a conservative way, the controller first adds a filtering rule on s_1 at t_1, then removes the filtering rule on s_3 at t_2, after the first rule addition has been confirmed.

The forwarding graphs at all steps seem correct. However, if a packet enters s_1 before t_1 and reaches s_3 after t_2, it will reach s_4, which violates the policy. Traversal of a packet over the network is not atomic, interleaving with network updates, as also observed in [63]. Moreover, [**?**] recently proved that there are situations where no correct update order exists. To deal with it, upon receiving an ack from the network, *CCG* does not immediately mark the state of the corresponding forwarding link as certain. Instead, it delays application of the confirmation to its internal data structure. In fact, confirmations of additions of forwarding links in the graph model can be processed immediately, and only confirmations of removals of forwarding links need to be delayed. The reason is that we want to ensure we represent all the possible behaviors of the network. Even after a forwarding rule has been deleted, packets processed by the rule may still exist in the network, buffered in an output queue of that device, in flight, or on other devices.

We have proved that our uncertainty-aware model is able to accurately capture the view of the network from the packets' perspective A, even for in-flight packets that have been affected by rules not currently present.

Definition 3.1. *A packet P's view of the network* **agrees** *with the uncertainty-aware model, if at any time point during its traversal of the network, the data plane state that the packet encounters is in the model at that time point. More specifically, at time t, to P if a link l*

- *is reachable, l is in the graph model for P at t;*
- *otherwise, l is definitely not certain in the graph at t.*

Theorem 3.1. *Assuming that all data plane changes are initiated by the controller, any packet's view of the network agrees with the uncertainty-aware model.*

3.3.4 Uncertainty-aware Verification

Construction of a *correct* network verification tool is straightforward with our uncertainty-aware model. By traversing the uncertainty graph model using directed graph algorithms, we can answer queries such as whether a reachable path exists between a pair of nodes. That can be done in a manner similar to existing network verification tools like HSA [39] and VeriFlow [43]. However, the traversal process needs to be modified to take into account uncertainty. When traversing an uncertain link, we need to keep track of the fact that downstream inferences lack certainty. If we reach a node with no *certain* outgoing links, it is possible that packets will encounter a black-hole even with multiple *uncertain* outgoing links available. By traversing the graph once, *CCG* can reason about the network state correctly in the presence of uncertainty, determine if an invariant is violated, and output the set of possible conterexamples (e.g., a packet and the forwarding table entries that caused the problem).

3.4 Consistency under Uncertainty

In this section, we describe how we use our model to efficiently synthesize update sequences that obey a set of provided invariants (§3.4.1). We then identify a class of invariants that can be guaranteed in this manner (§3.4.2), and present our technique to preserve consistency for broader types of invariants (§3.4.3).

3.4.1 Enforcing Correctness with Greedily Maximized Parallelism

The key goal of our system is to instill user-specified notions of correctness during network transitions. The basic idea is relatively straightforward. We construct a *buffer* of updates received from the application, and attempt to send them out in FIFO order. Before each update is sent,

we check with the verification engine on whether there is any possibility, given the uncertainty in network state, that sending it could result in an invariant violation. If so, the update remains buffered until it is safe to be sent.

There are two key problems with this approach. The first is head-of-line blocking: it may be safe to send an update, but one before it in the queue, which isn't safe, could block it. This introduces additional delays in propagating updates. Second, only one update is sent at a time, which is wasteful—if groups of updates do not conflict with each other, they could be sent in parallel.

To address this, *CCG* provides an algorithm for synthesizing update sequences to networks that greedily *maximizes parallelism* while simultaneously obeying the supplied properties (Algorithm 1).

Whenever an update u is issued from the controller, *CCG* intercepts it before it hits the network. Network forwarding behavior is modeled as an uncertainty graph ($G_{uncertain}$) as described previously. Next, the black-box verification engine takes the graph and the new update as input, and performs a computation to determine whether there is any possibility that the update will cause the graph state to violate any policy internally specified within this engine. If the verification is passed, the update u is sent to the network and also applied to the network model $Model$, but marked as uncertain. Otherwise, the update is buffered temporarily in Buf.

When a confirmation of u from the network arrives, *CCG* also intercepts it. The status of u in $Model$ is changed to certain, either immediately (if u doesn't remove any forwarding link from the graph), or after a delay (if it does, as described in §3.3.3). The status change of u may allow some pending updates that previously failed the verification to pass it. Each of the buffered updates is processed through the routine of processing a new update, as described above.

In this way, *CCG* maintains the order of updates only when it matters. Take the example in Figure 3.2. If the deletion of rule 1 is issued before the addition of rule 2 is confirmed, *CCG*'s verification engine will capture a possible loop, and thus will buffer the deletion update. Once the confirmation of adding rule 2 arrives, *CCG* checks buffered updates, and finds out that now it's safe to issue the deletion instruction.

Algorithm 3.1 Maximizing network update parallelism

ScheduleIndividualUpdate($Model, Buf, u$)

On issuing u**:**
$G_{uncertain}$ = **ExtractGraph**($Model, u$)
$verify$ = **BlackboxVerification**($G_{uncertain}, u$)
if $verify$ == **PASS then**
    Issue u
    Update($Model, u, uncertain$)
else
    Buffer u **in** Buf

On confirming u**:**
Update($Model, u, certain$)
$Issue\_updates \leftarrow \emptyset$
for $u_b \in Buf$ **do**
    $G_{uncertain}$ = **ExtractGraph**($Model, u_b$)
    $verify$ = **BlackboxVerification**($G_{uncertain}, u_b$)
    if $verify$ == **PASS then**
        Remove u_b **from** Buf
        Update($Model, u_b, uncertain$)
        $Issue\_updates \leftarrow Issue\_updates + u_b$
Issue $Issue\_updates$

3.4.2 Segment Independence

Next, we identify a class of invariants for which a feasible update ordering exists, and for which *CCG*'s heuristic will be guaranteed to find one such order. As defined in [63], *trace properties* characterize the paths that packets traverse through the network. This covers many common network properties, including reachability, access control, loop freedom, and waypointing. We start with the assumption that a network configuration applies to exactly one equivalence class of packets. A network configuration can be expressed as a set of paths that packets are allowed to take, i.e., a forwarding graph. A configuration transition is equivalent to a transition from an initial forwarding graph, G_0, to a final graph, G_f, through a series of transient graphs, G_t, for $t \in \{1, \ldots, f-1\}$. We assume throughout that the invariant of interest is preserved in G_0 and G_f.

Loop and black-hole freedom The following theorems were proved for loop freedom [28]: First, given both G_0 and G_f are loop-free, during transition, it is safe (causing no loop) to update a node in any G_t, if that node satisfies one of the following two conditions: (1) in G_t it is a leaf node, or all its upstream nodes have been updated with respect to G_f; or (2) in G_f it reaches the destination directly, or all its downstream nodes in G_f have been updated with respect to G_f. Second, if there

are several updatable nodes in a G_t, any update order among these nodes is loop-free. Third, in any loop-free G_t (including G_0) that is not G_f, there is at least one node safe to update, i.e., a loop-free update order always exists.

Similarly, we have the following proved for the black-hole freedom property [4].

Lemma 3.1. *(Updatable condition): A node update does not cause any transient black-hole, if in G_f, the node reaches the destination directly, or in G_t, all its downstream nodes in G_f have already been updated.*

Proof. By contradiction. Let N_0, N_1,...N_n be downstream nodes of N_a in G_f. Assume N_0, N_1,...N_n have been updated with respect to G_f in G_t. After updating N_a in G_t, N_0, N_1,...N_n become N_a's downstream nodes and all nodes in the chain from N_a to N_n have been updated. N_a's upstream with respect to G_t can still reach N_a, and thus reach the downstream of N_a. If we assume there is a black-hole from updating N_a, there exists a black-hole in the chain from N_a to N_n. Therefore, the black-hole will exist in G_f, and there is a contradiction. □

Lemma 3.2. *(Simultaneous updates): Starting with any G_t, any update order among updatable nodes is black-hole-free.*

Proof. Consider a updatable node N_a such that all its downstream nodes in G_f have already been updated in G_t (Lemma 1). Then updating any other updatable node does not change this property. When a node is updatable it remains updatable even after updating other nodes. Therefore, if there are several updatable nodes, they can be updated in any order or simultaneously. □

Theorem 3.2. *(Existence of a black-hole-free update order): In any black-hole-free G_t that is not G_f (including G_0), at least one of the nodes is updatable, i.e., there is a black-hole-free update order.*

Proof. By contradiction. Assume there is a transient graph G_t such that no node is updatable. All nodes are either updated or not updatable. As nodes with direct links to the destination are updatable (Lemma 1), these nodes can only be updated. Then nodes at previous hop of these nodes in G_t are also updatable (Lemma 1), and therefore these nodes must also be updated. Continuing, it follows that all nodes are updated, which is a contradiction as $G_t = G_f$. As there is always a

node updatable in a consistent G_t, and the updatable node can be updated to form a new consistent G_t, the number of updated nodes will increase. Eventually, all nodes will be updated. Therefore there is a black-hole free update order. □

Any update approved by *CCG* results in a consistent transient graph, so *CCG* always finds a consistent update sequence to ensure loop and black-hole freedom.

Generalized Trace Properties To get a uniform abstraction for trace properties, let us first visit the basic connectivity problem: node A should reach node B ($A \rightarrow B$). To make sure there is connectivity between two nodes, both black-hole and loop freedom properties need to hold. Obviously, black-hole freedom is downstream-dependent (Theorem 2), whereas loop freedom is upstream- (updatable condition (1)) **or** downstream-dependent (updatable condition (2)), and thus weaker than black-hole freedom. In other words, connectivity is a downstream-dependent property, i.e., updating from downstream to upstream is sufficient to ensure it. Fortunately, a number of trace properties, such as waypointing, access control, service/middle box chaining, etc., can be broken down to basic connectivity problems. A common characteristic of such properties is that flows are required to traverse a set of waypoints.

Definition 3.2. Waypoints-based trace property: *A property that specifies that each packet should traverse a set of waypoints (including source and destination) in a particular order.*

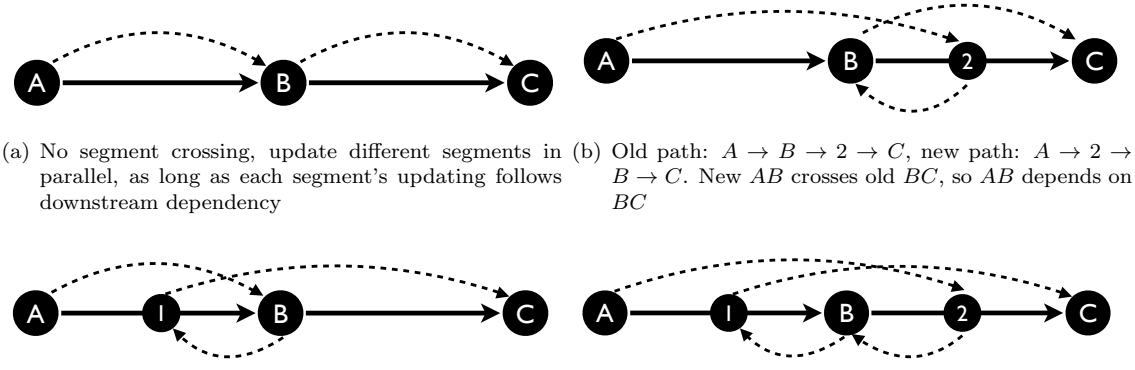
Definition 3.3. Segment dependency: *Suppose a trace property specifies n waypoints, which divide the old and the new flow path each into $(n-1)$ segments: $old_1, old_2, ..., old_{n-1}$ and $new_1, new_2, ..., new_{n-1}$. If new_j crosses old_i $(i \neq j)$, then the update of segment j is* **dependent** *on the update of segment i, i.e., segment j cannot start to update until segment i's update has finished, in order to ensure the traversal of all waypoints.*

Otherwise, if segment j starts to update before i has finished, there might be violations. If $j < i$, there might be a moment when the path between waypoints j and $i+1$ consists only of new_j and part of old_i, i.e., waypoints $(j+1)...i$ are skipped. As in Figure 3.5(b), B may be skipped if the AB segment is updated before BC, and the path is temporarily $A \rightarrow 2 \rightarrow C$.

If $j > i$, there might be a moment when the path between waypoints i and $(j+1)$ consists

of $old_i, old_{i+1}, ..., new_j$, and a loop is formed. As in Figure 3.5(c), the path could temporarily be $A \to B \to 1 \to B$.

If there is no dependency among segments (Figure 3.5 (a)), then each can be updated independently simply by ensuring connectivity between the segment's endpoints. That suggests that for paths with no inter-segment dependencies, a property-compliant update order always exists. Another special case is circular dependency between segments, as depicted in Figure 3.5(d), in which no feasible update order exists.



(a) No segment crossing, update different segments in parallel, as long as each segment's updating follows downstream dependency

(b) Old path: $A \to B \to 2 \to C$, new path: $A \to 2 \to B \to C$. New AB crosses old BC, so AB depends on BC

(c) Old path: $A \to 1 \to B \to C$, new path: $A \to B \to 1 \to C$. New BC crosses old AB, so BC depends on AB

(d) Old path: $A \to 1 \to B \to 2 \to C$, new path: $A \to 2 \to B \to 1 \to C$. New BC crosses old AB, and new AB crosses old BC, so BC and AB have circular dependency between themselves.

Figure 3.5: *Examples: dependencies between segments. Path AC is divided into two segments AB and BC by three waypoints A, B, and C, with old paths in solid lines, and new paths in dashed lines.*

Theorem 3.3. *If there is no circular dependency between segments, then an update order that preserves the required property always exists. In particular, if policies are enforcing no more than two waypoints, an update order always exists.*

If a policy introduces no circular dependency, i.e., at least one segment can be updated independently (Figure 3.5(a-c)), then we say the policy is *segment independent*. However, in reality, forwarding links and paths may be shared by different sets of packets, e.g., multiple flows. Thus it is possible that two forwarding links (smallest possible segments) l_1 and l_2 will have conflicting dependencies when serving different groups of packets, e.g., in forwarding graphs destined to two different IP prefixes. In such cases, circular dependencies are formed across forwarding graphs. Fortunately, forwarding graphs do not share links in many cases. For example, as pointed out

in [36], a number of flow-based traffic management applications for the network core (e.g., ElasticTree, MicroTE, B4, SWAN [18, 32, 34, 35]), any forwarding rule at a switch matches at most one flow.

Other Properties There are trace properties which are not waypoint-based, such as quantitative properties like path length constraint. To preserve such properties and waypoint-based trace properties that are not segment independent, we can use other heavyweight techniques as a fallback (see 3.4.3), such as CU [63]. Besides, there are network properties beyond trace properties, such as congestion freedom, and it has been proven that careful ordering of updates cannot always guarantee congestion freedom [34, 66]. To ensure congestion freedom, one approach is to use other heavyweight tools, such as SWAN [34], as a fallback mechanism that the default heuristic algorithm can trigger only when necessary.

3.4.3 Synthesis of Consistent Update Schedules

When desired policies do not have the segment-independence property (§3.4.2), it is possible that some buffered updates (through very rare in our experiments) never pass the verification. For instance, consider a circular network with three nodes, in which each node has two types of rules: one type to forward packets to destinations directly connected to itself, and one default rule, which covers destinations connected to the other two switches. Initially, default rules point clockwise. They later change to point counterclockwise. No matter which of the new default rules changes first, a loop is immediately caused for some destination. The loop freedom property is not segment-independent in this case, because each default rule is shared by two equivalence classes (destined to two hosts), which results in conflicting dependencies among forwarding links.

To handle such scenarios, we adopt a hybrid approach (Algorithm 2). If the network operators desire some policies that can be guaranteed by existing solutions, e.g., CU or SWAN, such solutions can be specified and plugged in as the fallback mechanism, FB. The stream of updates is first handled by CCG's greedy heuristic (Algorithm 1) as long as the policy is preserved. Updates that violate the policy are buffered temporarily. When the buffering time is over threshold T, configured by the operator, the fallback mechanism is triggered. The remaining updates are fed into FB to be transformed to a feasible sequence, and then Algorithm 1 proceeds with them again to heuristically

Algorithm 3.2 Synthesizing update orderings

ScheduleUpdates($Model, Buf, U, FB, T$)
  for $u \in U$ **do**
    ScheduleIndividualUpdate($Model, Buf, u$)

  On timeout(T):
  $\tilde{U}$ = **Translate**(Buf, FB)
  for $u \in \tilde{U}$ **do**
    ScheduleIndividualUpdate($Model, Buf, u$)

maximize update parallelism. In that way, *CCG* can always generate a consistent update sequence, *assuming a fallback mechanism exists which can guarantee the desired invariants.*[1] Note that even with *FB* triggered, *CCG* achieves better efficiency than using *FB* alone to update the network, because: 1) in the common case, most of updates are not handled by *FB*; 2) *CCG* only uses *FB* to "translate" buffered updates and then heuristically parallelize issuing the output of *FB*, but doesn't wait explicitly as some *FB* mechanism does, e.g., the waiting time between two phases in CU.

To show the feasibility of that approach, we implemented both CU [63] (see §3.6) and SWAN [34] as our fallback mechanisms in *CCG*. We emulated traffic engineering (TE) and failure recovery (FR), similar to Dionysus [36], in the network shown in Figure 3.6. Network updates were synthesized to preserve congestion-freeness using *CCG* (with SWAN as plug-in), and for comparison, using SWAN alone. In the TE case, we changed the network traffic to trigger new routing updates to match the traffic. In the FR case, we turned down the link S3-S8 so that link S1-S8 was overloaded. Then the FR application computed new updates to balance the traffic. The detailed events that occurred at all eight switches are depicted in Figure 3.7. We see that *CCG* ensured the same consistency level, but greatly enhanced parallelism, and thus achieved significant speed improvement (1.95× faster in the TE case, and 1.97× faster in the FR case).

[1] If no appropriate fallback exists, and the invariant is non-segment-independent, *CCG* can no longer guarantee the invariant. In this case, *CCG* can offer a "best effort" mechanism to maintain consistency during updates by simply releasing buffered updates to the network after a configurable threshold of time. This approach might even be preferable for certain invariants where operators highly value update efficiency; we leave an evaluation to future work.

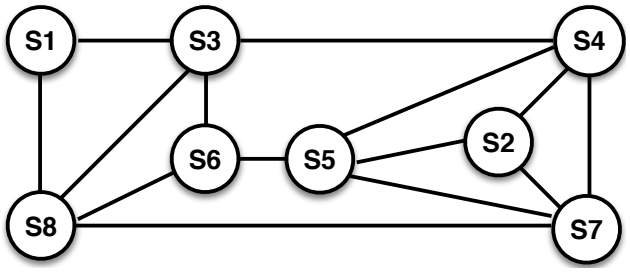


Figure 3.6: *Topology for CCG and SWAN bandwidth tests*

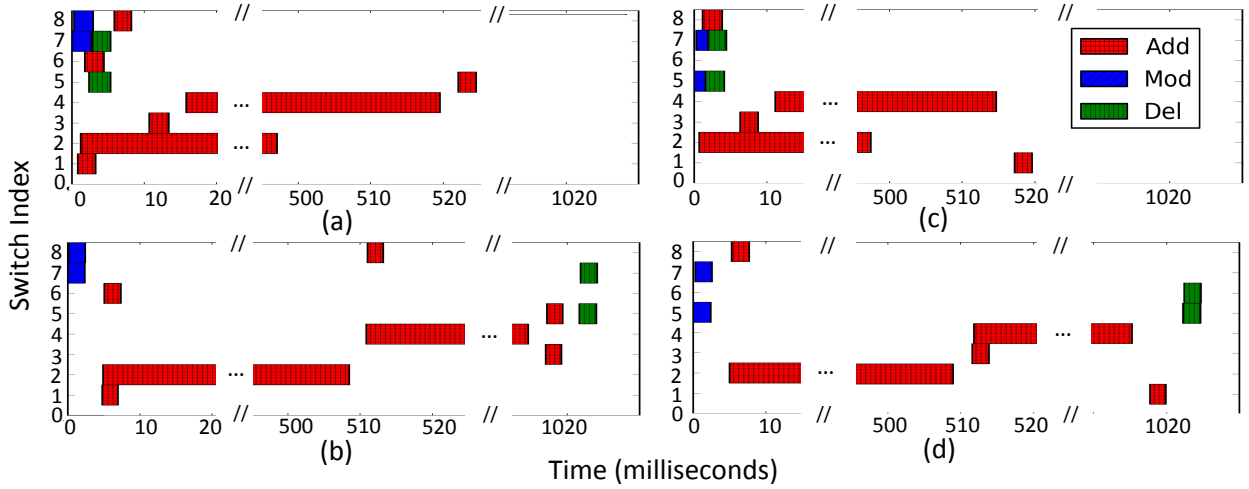


Figure 3.7: *Time series of events that occurred across all switches: (a) SWAN + CCG, traffic engineering; (b) SWAN, traffic engineering; (c) SWAN + CCG, failure recovery; (d) SWAN, failure recovery.*

3.5 Implementation

We describe four key aspects of our implementation: 1) how to efficiently construct and maintain the uncertainty-aware model, 2) how to bound temporary uncertainty, 3) how to accelerate verification with storage data structure optimization, and 4) how to support bandwidth checking.

3.5.1 Representing Network Uncertainty

In our uncertainty graph, we need to keep track of the state of each link. At first glance, we can do this by keeping a simple *state machine* for each link, which has two states: certain, and uncertain. However, two states are not sufficient, as there are some complexities which arise in practice. For example, there may be dependencies across updates – the question of whether to mark a link as certain can depend on updates that follow.

To address this, *CCG* keeps track of the state of each rule, and associate them each with a status of uncertainty as well as two counters. The status of uncertainty could be one in the set {*uncertain, certain, pre-certain, unknown*}. The counter, *add_cnt*, records the number of insertions of the rule, and the counter, *del_cnt* records the number of deletions.

Figure 3.8 shows the state machine of our uncertainty-aware model. The *certain* and *uncertain* states are consistent with the notion during modeling phase. The *pre-certain* and *unknown* require some additional explanation, which we will clarify through an example. Suppose that rule (update) R is issued by the controller. Before the controller receives an acknowledgement of R, the status of R in our storage data structure (discussed later in §3.5.3) is *uncertain* and its *add_cnt* counter increases from zero to one. Once R is confirmed, R's status is changed to *certain*, and the *add_cnt* counter is reset to zero. If another insertion of R is issued before the first R is confirmed, then the status of R remains *uncertain*, and the *add_cnt* counter is increased to two. Assume one confirmation is received now. We are certain that R exists in the network, but there is an in-flight control message to add R again, which may conflict with a later deletion. Such case is represented by the status *pre-certain*. Note that if a rule is *pre-certain* for adding action (*Add_pre_certain*), it is certainly included in the model. Similarly, if a rule is in the state *Del_pre_certain*, then the rule is not modelled. The status, *unknown*, describes the case that the same rule is first added and then deleted (or the other way round), but the controller has not yet received a confirmation of the first command. Hence, the order of applying the two in-flight commands is unknown, even after both commands are acknowledged.

3.5.2 Bounding Network Uncertainty

To bound the amount of time that the controller is uncertain about network states, a mechanism to acquire *confirmations* (an indication that the rule has been applied in the network) is required. Although existing SDN protocols such as OpenFlow do not support general application layer acknowledgments [55], there are other options, for example, (1) making use of the barrier and barrier reply messages of OpenFlow protocol, (2) leveraging an appropriately-chosen timeout or (3) having the controller actively query the data plane. We implemented two versions of the confirmation mechanism: an application-level acknowledgement by modifying the Stanford Open-

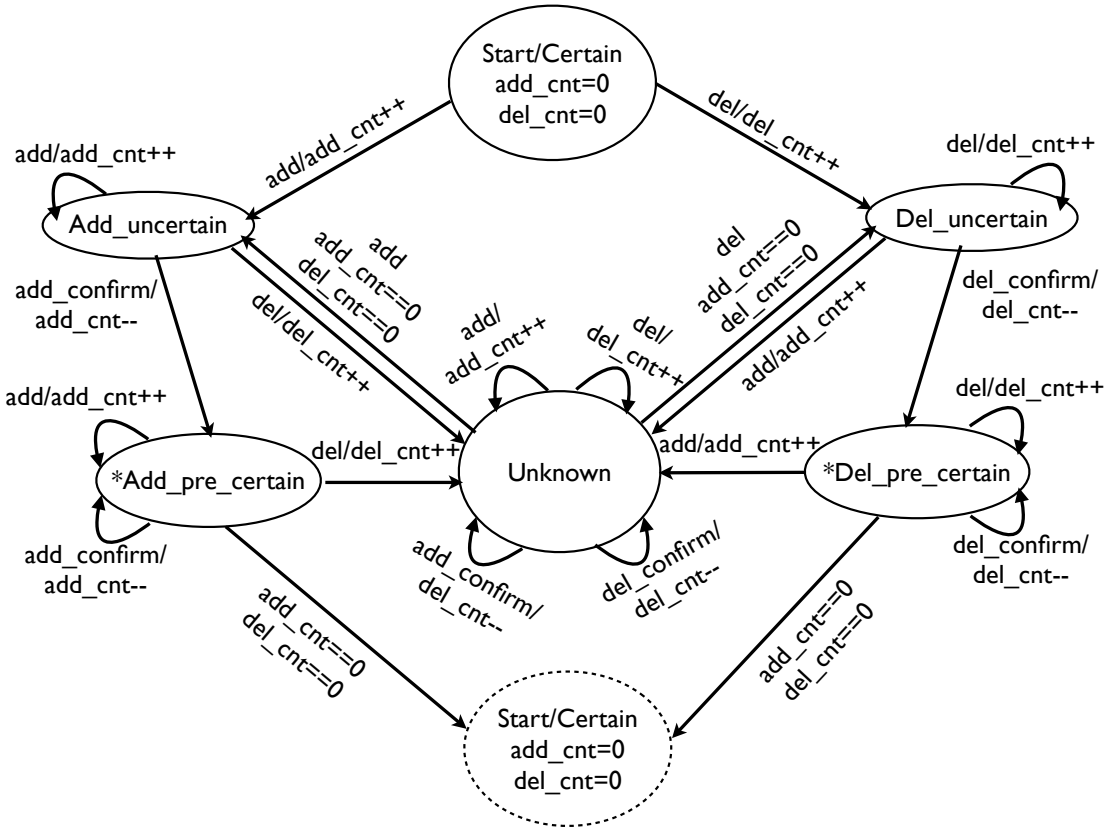


Figure 3.8: *Uncertainty state machine*

Flow switch reference implementation, (tested with a Mininet-based emulation), and a mechanism leveraging the barrier and barrier reply messages on our physical SDN testbed.

3.5.3 Efficiently Storing State

There are several pieces of state that *CCG* maintains, including network-wide data plane rules, uncertainty state of each rule (§3.5.1), and buffered updates. The key bottlenecks arise during the computationally-challenging procedure of storing and processing data plane rules.

The way that *CCG* stores and retrieves data plane rules is motivated by VeriFlow [43]. The essential idea is to build a multiple-layer trie, with each layer sub-trie representing a packet header field. Each level of a sub-trie corresponds to a bit of that field. An upper layer sub-trie contains pointers on leaves to sub-tries in the next layer. Dataplane rules are stored at the leaves of bottom sub-tries. A path from the root to a leaf of a bottom sub-trie determines a packet set, and one or more such sets can be merged together to form an *equivalence class* (EC) of packets, i.e., a set of

packets experiencing the same behaviors throughout the network. Upon arrival of an update, its effect on the network state is checked, by limiting searching to ECs whose behavior may be affected by this update, and building a graph model for each affected EC. However, given the need to store multiple different representations of the network state, the storage and processing overhead of *CCG* could be larger than just maintaining a single snapshot, as done in VeriFlow. Moreover, there are rules that perform packet transformation, such as Network Address Translation, i.e., these rules transform packets from one EC to another EC. To accurately model network behaviors, we need to able to handle such rules, which are left out in our motivating work, Veriflow. To meet this end, we design a scalable data structure and an efficient algorithm to operate on it.

Customized Trie Data Structure One complication is dealing with *wildcards* – rules agnostic to matching on certain bits or fields. Wildcards complicate the traversal process, as multiple branches may need to be traversed, and the manner in which they are traversed can depend on the semantics of the rule (e.g., standard wildcards vs. longest-prefix match). To address this, we construct an algorithm that handles general bitmasking, then extend it with optimizations to more efficiently handle two common-case wildcard patterns: full wildcards (the entire field is wildcarded) and subnet mask (all bits less than a certain significance are wildcarded). To deal with wildcarding for bitmasks, each node in our trie has three child branches, one for each of {0,1,don't care}. For subnetting, the wildcard branch has no children but points direct to a next layer sub-trie or a rule set. Thus, unlike other types of trie, the depth of subnet wildcard tries is not fixed as the number of bits in this field, but instead equals to the longest prefix among all the rules it stores. Accordingly, traversal cost is reduced compared with general tries. As for the full wildcard field, values can only be non-wildcarded or full wildcarded. The specialized trie structure for this type of field is a plain binary tree plus a wildcard table.

One-pass Traversal Algorithm When a new update arrives, we need to determine the set of affected ECs, as well as the rules affecting those ECs. VeriFlow [43] performs a similar task via a two-pass algorithm, first traversing the trie to compute a set of ECs, and then for each of the discovered ECs, traversing the trie again to extract related rules. In *CCG*, we optimize this process by maintaining some additional accounting information, which lets us accomplish our similar objective in a single pass. Our algorithm starts from the top layer subtrie, and combinations

of its branches that match the first field of the update to be checked are selected. The traversal continues on the matched combinations, and would be further confined by the following fields of the update. A matched combination of branches in the last level is an EC, and it already points to the rule set for that EC. Using callback functions and depth first searching, we implemented in place checking, and finish the modeling work with only one traversal. This algorithm eliminates both the unnecessary extra pass over the trie and also the need to allocate memory for intermediate results. In addition, this approach results in a smaller number of ECs—each wildcard branch itself forms a matched combination.

After getting the rule set for each EC, *CCG* traverses the graph model consisting of the collected rules, to check network invariants. One special case is dealing with packet transformation rules. When such a rule is encountered during graph traversal phase, a new pass over the trie for the transformed EC is triggered. Note that the graph is constructed while traversing rules, so only rules encountered before transformation are kept, and the remaining rule set is discarded. More importantly, in this case, multiple trie traversals are possible, but only when necessary, i.e., when a transformation rule is possibly forwarding packets for the original EC.

These efforts together with a highly optimized implementation allow *CCG* to run almost 100X faster compared to VeriFlow with 15X less memory overhead(§3.6.1). One of our ongoing works is exploring the parallel implementation of the data structure.

3.5.4 Checking Bandwidth Properties

Thanks to *CCG*'s graph-based model, it is straightforward to incorporate bandwidth information to the model. To keep tract of bandwidth usage, besides forwarding graphs, *CCG* also maintains a network-wide physical topology graph. Whenever a set of rules are issued to set up paths for flows with bandwidth requirement, *CCG* reserves bandwidth on all possible paths taken by those flows on the physical graph model. As rules are confirmed to be removed, *CCG* releases bandwidth occupied by flows that these rules used to forward. Results shown previously (§3.4.3) on the integration of *CCG* and SWAN proves the feasibility of this implementation.

3.6 Evaluation

3.6.1 Verification Time

To gain a baseline understanding of *CCG*'s performance, we micro-benchmarked how long the verification engine takes to verify a single update. We simulated BGP routing changes by replaying traces collected from the Route Views Project [15], on a network consisting of 172 routers following a Rocketfuel topology (AS 1755) [12]. After initializing the network with 90,000 BGP updates, 2,559,251 updates were fed into *CCG* and VeriFlow [43] (as comparison). We also varied the number of concurrent uncertain rules in *CCG* from 100 to 10,000. All experiments were performed on a 12-core machine with Intel Core i7 CPU at 3.33 GHz, and 18 GB of RAM, running 64-bit Ubuntu Linux 12.04. The CDFs of the update verification time are shown in Figure 3.9.

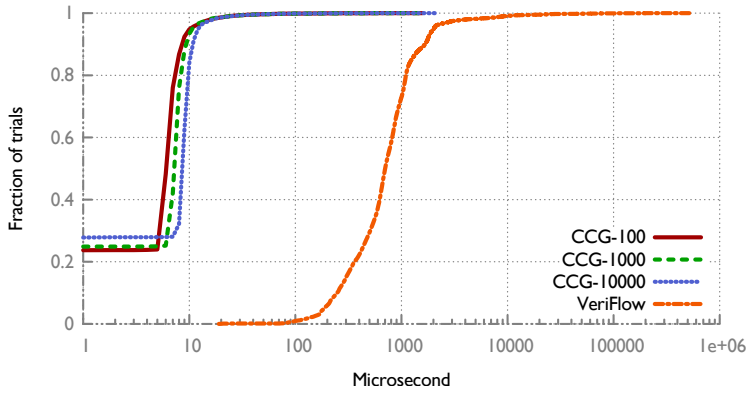


Figure 3.9: *Microbenchmark results.*

CCG was able to verify 80% of the updates within 10 μs, with a 9 μs mean. *CCG* verifies updates almost two order of magnitude faster than VeriFlow because of data structure optimizations (§3.5). Approximately 25% of the updates were processed within 1 μs, because *CCG* accurately tracks the state of each rule over time. When a new update matches the pattern of some existing rule, it's likely only a minimum change to *CCG*'s network model is required (e.g., only one operation in the trie, with no unnecessary verification triggered). We observed long tails in all curves, but the verification time of *CCG* is bounded by 2.16 ms, almost three orders of magnitude faster than VeriFlow's worst case. The results also show strong scalability. As the number of concurrent uncertainty rules grows, the verification time increases slightly (on average, 6.6 μs, 7.3 μs, and 8.2 μs for the 100-, 1000-, and 10000-uncertain-rule cases, respectively). Moreover, *CCG* offers a

significant memory overhead reduction relative to VeriFlow: 540 MB vs 9 GB.

3.6.2 Update Performance Analysis

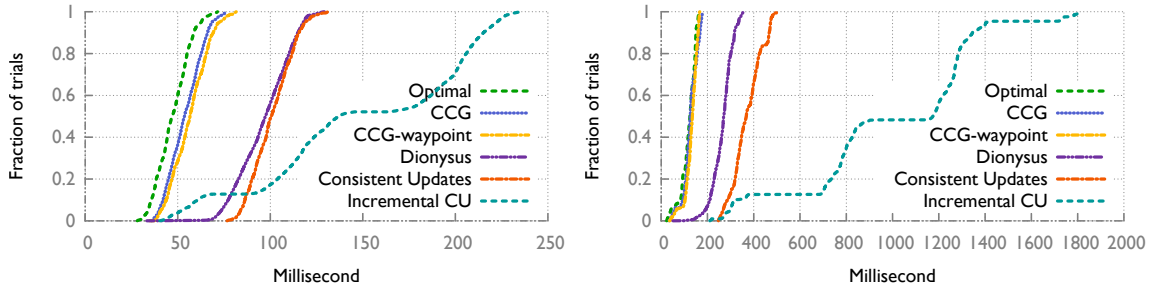
Emulation-based Evaluation

Segment-independent Policies: We used Mininet to emulate a fat-tree network with a shortest path routing application and a load-balancing application in a NOX controller. The network consists of five core switches and ten edge switches, and each edge switch connects to five hosts. We change the network (e.g., add links, or migrate hosts) to trigger the controller to update the data plane with a set of new updates. For each set of experiments, we tested six update mechanisms: (1) the controller immediately issues updates to the network, which is **Optimal** in terms of update speed; (2) *CCG* with the basic connectivity invariants, loop and black-hole freedom, enabled (*CCG*); (3) *CCG* with an additional invariant that packets must traverse a specific middle hop before reaching the destination (*CCG*-waypoint); (4) Consistent Updates (**CU**) [63]; (5) incremental Consistent Updates (**Incremental CU**) [38]; and (6) **Dionysus** [36] with its WCMP forwarding dependency graph generator. We configure our applications as the same type as in Dionysus, with forwarding rules matching exactly one flow, i.e., no overlapping forwarding graphs. Thus, loop and black-hole freedom are segment-independent as proved in §3.4.2. Because of the fat-tree structure, there is no crossing between path segments (as in Fig 3.5(a)), so the waypoint policy is also segment independent. A mix of old and new configurations, e.g., $oldAB + newBC$ in Figure 3.5(a), is allowed by *CCG*, but forbidden when using CU. Note here, we used our own implementation of the algorithms introduced in Dionysus paper, specifically the algorithm for packet coherence. Therefore, this is not a full evaluation of the Dionysus *approach*: one can develop special-purpose algorithms that build customized dependency graphs for weaker properties, and thus achieve better efficiency. We leave such evaluation to future work.

We first set the delay between the controller issuing an update and the corresponding switch finishing the application of the update (i.e, the controller-switch delay) to a normal distribution with 4 ms mean and 3 ms jitter, to mimic a dynamic data center network environment. The settings are in line with that of other data center SDN experiments [23, 65]. We initialized the test with one core switch enabled and added the other four core switches after 10 seconds. The

traffic eventually is evenly distributed across all links because of the load balancer application. We measured the completion time of updating each communication path, repeated each experiment 10 times. Figure 3.10(a) shows the CDFs for all six scenarios.

The performance of both "*CCG*" and "*CCG*-waypoint" is close to optimal, and much faster (47 ms reduction on average) than CU. In CU, the controller is required to wait for the maximum controller-switch delay to guarantee that all packets can only be handled by either the old or the new rules. *CCG* relaxes the constraints by allowing a packet being handled by a mixture of old and new rules along the paths, as long as the impact of the new rules passed verification. By doing so, *CCG* can apply any verified updates without explicitly waiting for irrelevant updates. CU requires temporary doubling of the FIB space for each update, because it does not delete old rules until all in-flight packets processed by the old configuration have drained out of the network. To address this, incremental-CU was proposed to trade time against flow table space. By breaking a batch of updates into k subgroups ($k = 3$ in our tests), incremental-CU reduced the extra memory usage to roughly one kth at the cost of multiplying the update time k times. In contrast, when dealing with segment-independent policies, as in this set of experiments, *CCG* never needs to trigger any heavyweight fallback plug-in, and thus requires no additional memory, which is particularly useful as switch TCAM memory can be expensive and power-hungry.



(a) Data center network setting

(b) Wide-area network setting

Figure 3.10: *Emulation results: update completion time comparison.*

To understand how *CCG* performs in wide-area networks, where SDNs have also been used [34, 35], we set the controller-switch delay to 100 ms (normal distribution, with 25ms jitter), and repeated the same tests (Figure 3.10(b)). *CCG* saved over 200 ms update completion time com-

pared to CU, mainly due to the longer controller-switch delay, for which CU and incremental-CU have to wait between the two phases of updates.

As for Dionysus, we observed in Figure 3.10 that it speeds up updates compared to CU in both local and wide-area settings, as it reacts to network dynamics rather than pre-determining a schedule. But because its default algorithm for WCMP forwarding produces basically the same number of updates as CU, *CCG* (either *CCG* or *CCG*-waypoint) outperforms it in both time and memory cost. We further compared *CCG*-waypoint with Dionysus in other dynamic situations, by varying controller-switch delay distribution. Figure 3.11 shows the 50^{th}, 90^{th} and 99^{th} percentile update completion time, under various controller-switch delays (normal distributed with different (mean, jitter) pairs, (a, b)) for four update mechanisms: optimal, *CCG*, Dionysus, and CU. In most cases, both *CCG* and Dionysus outperform CU, with one exception (4ms delay, zero jitter). Here, Dionysus does not outperform CU because it adjusts its schedule according to network dynamics, which was almost absent in this scenario. The cost of updating dependency graphs in this scenario is relatively large compared to the small network delay. When the mean delay was larger (100ms), even with no jitter, Dionysus managed to speed the transition by updating each forwarding path independently. On the other hand, *CCG*'s performance is closer to Optimal than Dionysus. For example, in the $(4, 0)$ case, *CCG* is 37%, 38%, and 52% faster than Dionysus in the 50^{th}, 90^{th} and 99^{th} percentile, respectively; in the $(100, 25)$ case, *CCG* is 50%, 50%, and 53% faster than Dionysus in the 50^{th}, 90^{th} and 99^{th} percentile, respectively. Also, we observe that Dionysus's performance is highly dependent on the variance of the controller-switch delay (the larger the jitter is, the faster the update speed) because of the dynamic scheduling, but *CCG*'s performance is insensitive to the jitter.

Non-segment-independent Policies: We then explored scenarios in which *CCG*'s lightweight heuristic cannot always synthesize a correct update ordering and needs to fall back to the more heavyweight algorithm to guarantee consistency. The traces we used were collected from a relatively large enterprise network that consists of over 200 layer-3 devices. During a one-day period (from 16:00 7/22/2014 to 16:00 7/23/2014), we took one snapshot of the network per hour, and used Mininet to emulate 24 transitions, each between two successive snapshots. We processed the network updates with three mechanisms: immediate application of updates, *CCG*, and CU.

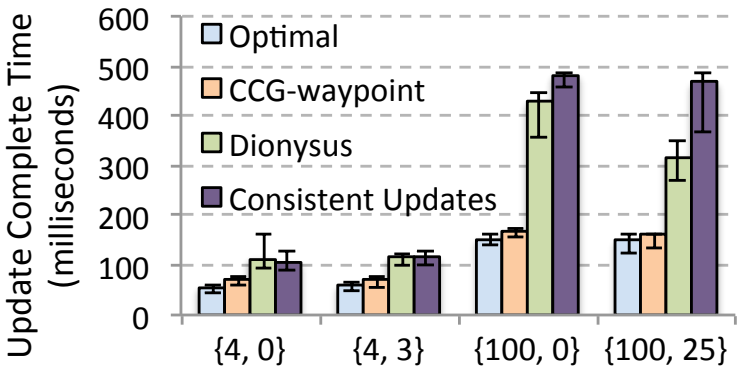


Figure 3.11: *Update completion time with [50^{th}, 90^{th}, 99^{th} percentile]; x-axis label {a, b}: a is the mean controller-switch delay, b is the jitter following a normal distribution.*

Updates were issued such that new rules were added first, then old rules deleted. Thus, all three mechanisms experience the trend that the number of stored rules increases then decreases.. The controller-switch delay was set to 4 ms. We selected 10 strongly connected devices in the network, and plotted the number of rules in the network over time during four transition windows, as shown in Figure 3.12. As the collected rules overlapped with longest prefix match, the resulting forwarding graphs might share links, so unlike previous experiments, segment-independence was not guaranteed.

The update completion time (indicated by the width of the span of each curve) using *CCG* was much shorter than CU, and the memory needed to store the rules was much smaller. In fact, the speed and memory requirements of *CCG* were close to those of the immediate update case, because *CCG* rarely needs to fall back to CU. In 22 out of 24 windows, there was a relatively small number of network updates (around 100+), much as in the [22:00, 23:00) window shown in Figure 3.12, in which *CCG* passed through most of the updates with very few fallbacks. During the period 23:00 to 1:00, there was a burst of network dynamics (likely to have been caused by network maintenance), in which 8000+ network updates occurred. Even for such a large number of updates, the number of updates forced to a fallback to CU, was still quite small (10+). Since *CCG* only schedules updates in a heuristic way, the waiting time of a buffered update could be suboptimal, as in this hour's case, where the final completion time of *CCG* was closer to CU. *CCG* achieves performance comparable to the immediate update mechanism, but without any of its short-term network faults (24 errors in the 0:00 to 2:00 period).

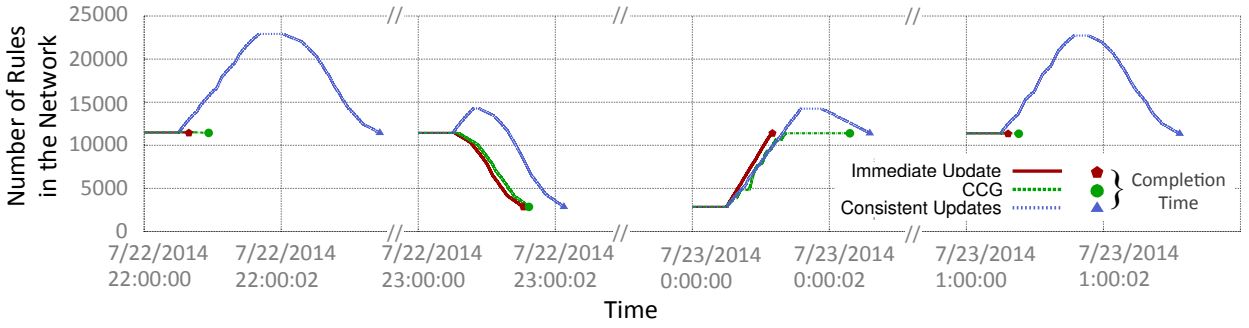


Figure 3.12: *Network-trace-driven emulations: (1) immediate application of updates; (2) CCG (with CU as fallback); and (3) CU.*

Physical-testbed-based Evaluation

We also evaluated *CCG* on a physical SDN testbed [14] consisting of 176 server ports and 676 switch ports, using Pica8 Pronto 3290 switches via TAM Networks, NIAGARA 32066 NICs from Interface Masters, and servers from Dell. We compared the performance of *CCG* and CU by monitoring the traffic throughput during network transitions. We first created a network with two sender-receiver pairs transmitting TCP traffic on gigabit links, shown in Figure 3.13. Initially, a single link was shared by the pairs, and two flows competed for bandwidth. After 90 seconds, another path was added (the upper portion with dashed lines in Figure 3.13). Eventually, one flow was migrated to the new path and each link was saturated. We repeated the experiment 10 times, and recorded the average throughput in a 100-ms window during the network changes. We observed repeatable results. Figure 3.14(a) shows the aggregated throughput over time for one trial.

CCG took 0.3 seconds less to finish the transition than CU because: (1) unlike CU, *CCG* does not require packet modification to support versioning, which takes on the order of microseconds for gigabit links, while packet forwarding is on the order of nanoseconds; (2) CU requires more rule updates and storage than *CCG*, and the speed of rule installation is around 200 flows per second; and (3) Pica8 OpenFlow switches (with firmware 1.6) cannot simultaneously process rule installations and packets.[2]

To test *CCG* in a larger setting, we then utilized all 13 physical switches. Each physical switch was divided into 6 "virtual" switches by creating 6 bridges. Due to the fact that the

[2]All the performance specifications reported here have been confirmed with the Pica8 technical team.

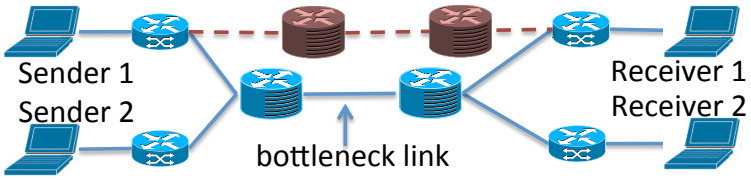
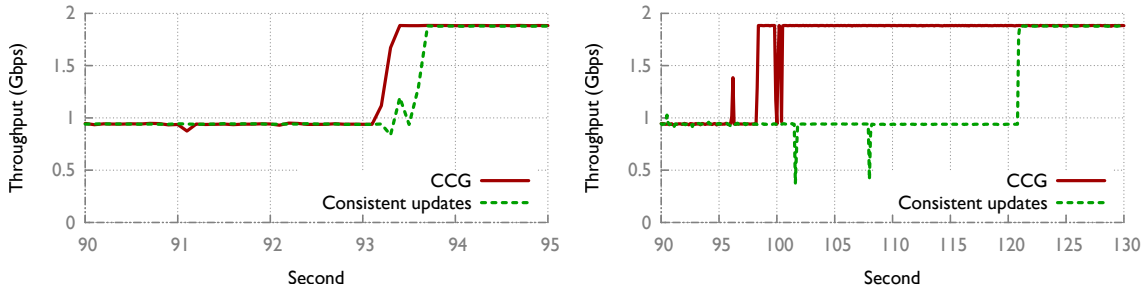


Figure 3.13: *eight-switch topology.*



(a) A eight-switch topology.

(b) A 78-switch network.

Figure 3.14: *Physical testbed results: comparison of throughput changes during network transitions for CCG and CU.*

switches are physically randomly connected, this division results in a "pseudo-random" network consisting of 78 switches, each with 8 ports. Initially, the topology consisted of 60 switches, and we randomly selected 10 sender-receiver pairs to transmit TCP traffic. After 90 seconds, we enabled the remaining 18 switches in the network. The topology change triggered installations of new rules to balance load. We repeated the experiments 10 times, and selected two flows from one trial that experienced throughput changes (Figure 3.14(b)). The trend of the two flows is consistent with the overall observed throughput change.

CCG again outperformed CU in convergence time and average throughput during transitions. Compared to CU, *CCG* spent 20 fewer seconds to complete the transition (a reduction of 2/3), because CU waits for confirmation of all updates in the first phase before proceeding to the second. In contrast, *CCG*'s algorithm significantly shortened the delay, especially for networks experiencing a large number of state changes. In *CCG*, the throughput never dropped below 0.9 Gb/s, while CU experienced temporary yet significant drops during the transition, primarily due to the switches' lack of support for simultaneous application of updates and processing of packets.

Table 3.1: Error Coverage Comparison: VeriFlow vs *CCG*

Error Type	Found by Veriflow & *CCG*	Only *CCG* (Potential)	Only *CCG* (Certain)
Black Hole	1,037,866	362,460	125,808
Loop	28,936	166,508	166,991
Out of Order Updates	N.A.	362,408	N.A.
Total	1,066,802	891,376	292,799

3.6.3 Network Fault Detection Coverage

Failing to consider the temporal uncertainty of the network may result in transient or permanent network faults that can affect security and performance. In this set of experiments, we explore how *CCG* can help to improve the error detection coverage by modeling network uncertainty. We used the same network topology consisting of that we used for the speed analysis in § 3.6.1. We replayed 2,559,251 BGP FIB changes into *CCG* and VeriFlow, and verified the network against forwarding loops and blackholes using both systems. Results are shown in Table 3.1.

The second column shows the number of errors that are detected by both VeriFlow and *CCG*. We measure three types of errors: black holes, loops, and out of order updates – the last of which refers to updates that trigger a race condition on switches, e.g., addition and withdrawal of the same rule without a barrier between them. We found that *CCG* does not miss any errors that VeriFlow captures. The third and fourth column show the number of errors that *CCG* is able to capture, but are missed by VeriFlow, among which 891,376 errors are reported by *CCG*'s verification engine as errors that may exist, and 292,799 as certain errors.

3.7 Summary

We present *CCG*, a system that enforces customizable network consistency properties with high efficiency. We highlight the network uncertainty problem and its ramifications, and propose a network modeling technique correctly derives consistent outputs even in the presence of uncertainty. The core algorithm of *CCG* leverages the uncertainty-aware network model, and synthesizes a feasible network update plan (ordering and timing of control messages). In addition to ensuring that there are no violations of consistency requirements, *CCG* also tries to maximize update parallelism, subject to the constraints imposed by the requirements. Through emulations and experiments on

an SDN testbed, we show that *CCG* is capable of achieving a better consistency vs. efficiency trade-off than existing mechanisms.

Chapter 4

Synthesizing Update Content

In this chapter, we will discuss the design of the component that synthesizes network update content, **N**etwork **E**rror **A**uto-Correc**t** (*NEAt*). Modern enterprise networks must comply with highly stringent security demands, including regulatory requirements, or industry standards, such as PCI, HIPAA, and SOX. As a result, network administrators must carefully design and maintain their networks to follow those policies, by mapping out device contexts and access to sensitive resources, assessing risk, and installing access control policies that effectively mitigate that risk. However, mistakes and errors in implementing the policies can result in costly data breaches, segmentation violations, and infiltrations. Through 2020, Gartner predicts 99% of firewall breaches will be caused by misconfigurations [1, 2].

While discovering and troubleshooting these bugs is essential to maintaining network security, doing so is notoriously hard. Relying on humans to configure and maintain the network configuration is not only prone to mistakes, but slow. Given the sophistication and speed at which new attack vectors propagate, manually updating and testing new configurations leaves the network in a vulnerable state until the attack vector is fully secured. Further, maintaining a security posture in the presence of software-defined networking (SDN) is even more challenging. While SDN enables new functionality, application designers may not be aware of the policy or security requirements of the networks on which their applications will be deployed. Worse yet, SDN applications written in general-purpose languages such as Java or Python can be arbitrarily complex. Requiring applications to implement and modify their behavior to support a broad spectrum of policies needed across a broad spectrum of networks presents an almost insurmountable challenge.

To this end, we present *NEAt*, a transparent layer to automatically repair policy-violating updates in real-time. *NEAt* secures the network with a mechanism similar to a smartphone's autocorrect feature, which enables on-the-fly repair to policy violating updates and ensures the network is always in a state consistent with policy. Unlike prior work on update synthesis, *NEAt*

maintains backward compatibility and flexibility to run general SDN application code. To do this, *NEAt* does not synthesize network state from scratch, but rather *influences* updates from an existing SDN application toward a correct specification. In particular, *NEAt* enforces a concrete definition of correctness by influencing and constraining dynamically arriving network instructions. To formulate those correctness criteria, we construct a set of *policy graphs* to represent humans' correctness intent, which is based on the observation that important error conditions can be caught by a concise set of boundary conditions. *NEAt* sits between an SDN controller and the forwarding devices, and intercepts the updates proposed by the running SDN applications. If the update violates an administrator's defined policy, such as reachability or segmentation, *NEAt* transforms the update into one that complies with the policy.

A key challenge we face in this approach is discovering update repairs in real-time. In *NEAt*, we build on prior work on verification to efficiently model packet forwarding behavior as a set of Equivalence Classes (ECs) [43, 72]. Upon receiving an update from an SDN controller, *NEAt* computes the set of affected ECs and checks for a violation in the same manner as [43]. To repair the violation, we cast the problem as an optimization problem, to find the minimum number of changes (added or deleted edges) to repair the violating EC's forwarding graph. To rapidly compute repairs on arbitrarily large networks, we exploit two optimization techniques, *topology limitation* which "slices" away irrelevant part of the network, and *graph compression*, to compress both an EC's forwarding graph and the topology. Then we solve the optimization problem on the sliced and compressed graphs.

Furthermore, as *NEAt* repairs policy-violating updates, stateful applications — without knowledge of the violating or repaired updates — will diverge from the underlying network state. To address this problem, applications can interactively propose updates to *NEAt* and receive notifications of repairs with minor modifications to application code. Thus, applications can remain unmodified and leverage *NEAt* transparently in a *pass-through* mode, with a risk of state divergence, or propose updates in an *interactive* mode.

A preliminary evaluation of our prototype shows promising results. On topologies with up to 400+ switches and 600+ hosts, *NEAt* can discover repairs in under one second for applications with non-overlapping rules, and under two seconds for applications with more complex dependencies.

Furthermore, we find *NEAt* can verify and repair updates on realistic data planes. On a large enterprise network with 1M forwarding rules, *NEAt* discovered and repaired 28 loop violations. Simulations on this data set show *NEAt* can verify and repair reachability and loop freedom policies in under a second.

4.1 Background and Motivation

Enterprise network policies must compose together requirements from a variety of demands, such as government or industry regulations, to mitigate risk for attack vectors, and limit access to sensitive resources. As a result, network administrators must take into account complex, composed policies when configuring or updating a network.

Composing together such policies is a slow and often error-prone process for a human operator. The operator may introduce errors translating the demands into high-level policies, or translating the policies into low-level routing configurations. A recent study [44] found that operators make changes to their networks at least once per day, while more than 80% were concerned updates would break existing functionality unrelated to a given change. While tools [39, 43] exist to automatically discover misconfigurations in real-time, they offer the operator no guidance on how to repair the misconfiguration beyond the type of correctness property that is violated

Instead, a system to automatically repair updates, ensuring the network always remains consistent with the administrator's policy, can relieve a slow and error-prone process from the configuration process. If an update violates a given property in the network, a *repair* should fix the cause of the violation while maintaining the original purpose of the update. We argue a minimal change is best, to repair the update with the least number of added or removed edges. Furthermore, such a system should improve upon a manual effort with transparency in both architecture and performance. A system that requires hours or days to verify and repair a network is not useful if the process can be completed manually in just a few minutes. It should also not require modifying existing applications or redesigning infrastructure.

However, accomplishing this task in real-time is challenging due to the size of the network and the data plane state. To efficiently reason about the data plane, we build on previous work in verification [43, 72] that separates the forwarding behavior into Equivalence Classes (ECs) of

packets. All packets within an EC are forwarded in precisely the same manner. From each EC, we can extract a *configuration graph* that defines the forwarding behavior for packets within the EC. A repair for a given EC must then explore additions or deletions of links in the configuration graph. Finding a link addition requires examining the *topology graph* defined by the edges in the physical topology. To efficiently discover repairs, we propose two optimization techniques to compress the configuration and topology graphs, described in §4.5. We refer to the outcome of these techniques as the *compressed configuration graph* and *compressed topology graph*.

Finally, a repair system should ideally support both unmodified and modified applications. Burdening developers or operators with the task of modifying their applications support the latest extension is unreasonable, but those that want more specific control over the repair process should have the option to do so. A *pass-through* mode could allow an operator to transparently leverage automated repairs, without the need to modify existing applications, while an *interactive* mode could provide the application with finer-grained control over the choices of repairs. Furthermore, an interactive mode would allow the application to ensure its state is consistent with that of the network, as it can update its own state after choosing one of several potential repairs for the policy-violating update.

4.2 Design

At the core of *NEAt* is a verification and correction layer, which ensures only updates conforming to the network policies are sent onto the network. This layer receives updates from one of two integration modes with the SDN control infrastructure: *pass-through* and *interactive*.

4.2.1 Verification and Repair

NEAt's verification and correction engines ensure the network is always consistent with the defined policy. To start, *NEAt* takes as input a *policy graph* (①), which defines the network policies (e.g., reachability, segmentation, waypointing) in the form of a directed graph. Next, *NEAt* receives updates (e.g., flow modification messages) from the SDN control infrastructure. With each update (②), *NEAt* applies the change to a network model, from which the ECs affected by the update are computed. Using the policy graph, *NEAt* checks each affected EC in the network model for

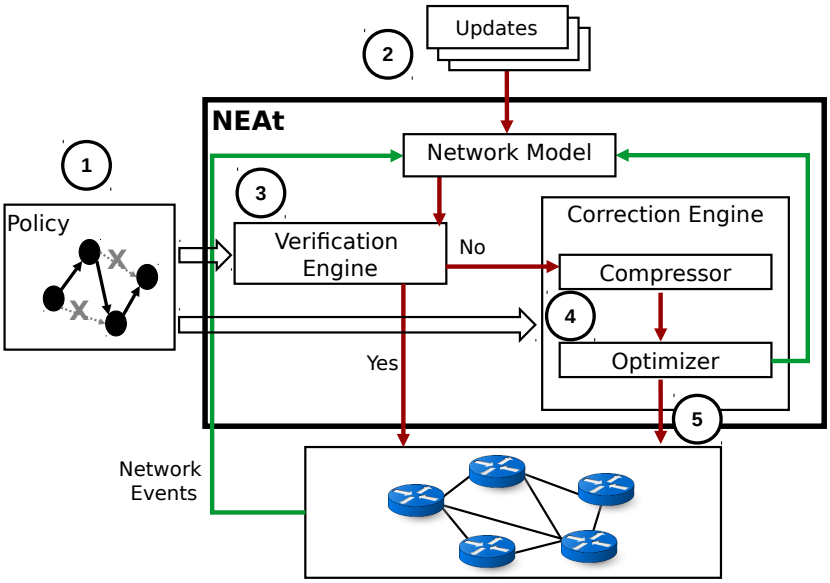


Figure 4.1: *System architecture of NEAt.*

policy violations using the verification engine (③). If the update does not introduce any violations, it is sent onto the network. However, if it does introduce a violation, the configuration graph and topology graph are compressed and passed to the correction engine (④). The optimizer returns a set of edges to be added or removed to the EC's configuration graph, which are then applied to the network model, converted to OpenFlow rules, and sent to the forwarding devices (⑤).

NEAt's correction engine models the process of discovering repairs as an optimization problem. Our exploration of alternative approaches guided us toward this optimization problem-based solution for performance considerations. For example, consider a brute force approach that discovers repairs for a given EC by testing all possible permutations of edge additions and removals to the EC's configuration graph. A repair that requires only adding edges, from 10 possible unused topology edges, would need to explore 10! (˜3.6M) permutations. If the violating property can be checked in just 1ms, each EC could take up to 10 minutes to find a repair. Therefore we use the formulation described in §4.4 for our repair discovery process.

4.2.2 Interaction Modes

With each repair, inconsistencies between application state and network state will arise. To prevent applications from diverging from the underlying network state, *NEAt* exposes two integration

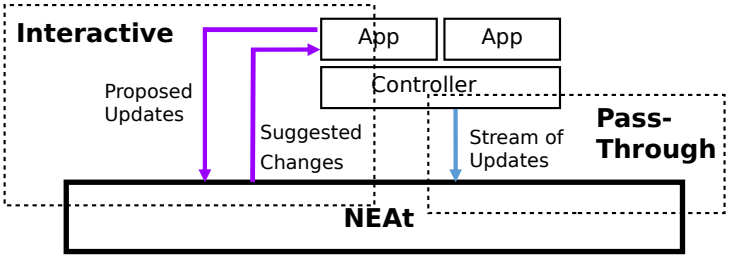


Figure 4.2: *Interaction modes of NEAt.*

modes: *pass-through* and *interactive*.

In pass-through mode, *NEAt* acts as a transparent layer that sits between the controller and forwarding devices. This mode enforces network policies without modifications to the controller applications. Both the controller and applications are unaware of *NEAt* in this mode. *NEAt* intercepts updates from the controller, as well as updates from the network about link and switch state, and passes it to the verification and repair engines.

Interactive mode enables applications to leverage *NEAt*'s verification and repair process by checking proposed updates. An application passes to *NEAt* a set of updates, which are checked against the current network model. If the updates introduce a violation, *NEAt* returns a set of repaired updates, which the application can accept or reject. If the application accepts the changes, it can send them onto the network and update its state, ensuring the application and network state are consistent. If the application rejects the changes, it can propose another set of updates to *NEAt*. Interactive mode requires modifications to applications to update its state with the accepted change.

NEAt maintains consistency between the interaction modes, allowing applications and the controller to both simultaneously benefit from *NEAt*'s automated repair. For example, one application can use *NEAt*'s API while another remains unmodified, allowing its updates to be checked by *NEAt* in pass-through mode.

4.3 Policy as Graphs

Many existing tools reason about individual network paths [41, 43]. While this approach has proven effective for network data plane verification, synthesizing network state changes requires

viewing the entire network as a whole (i.e., a graph), as changes that repair one path may influence the correctness of other paths. In addition, expressing network correctness conditions as a graph instead of a collection of paths enables dealing with a richer set of policies, for instance, path consistency and load balancing. Based on this intuition, *NEAt* takes as input a set of intended policies, and formulates these policies as directed graphs called *policy graphs*.

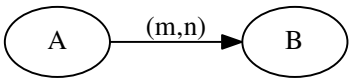


Figure 4.3: *Policy edge*

Edges on a *policy graph* are marked with notations denoting different types of reachability constraints. For example, the graph in Figure 4.3 specifies a requirement that at least m paths exist from node A to B, each bounded by n hops.

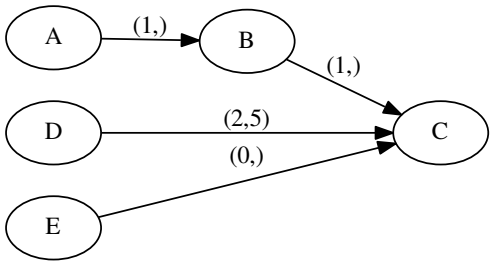


Figure 4.4: *Policy graph*

A basic *reachability* requirement can be expressed with $m = 1$ and n unspecified, as shown in edges $A \rightarrow B$ and $B \rightarrow C$ in Figure 4.4. When n is specified, we get a *bounded path length reachability* policy (e.g., $D \rightarrow C$ in Figure 4.4). *Shortest path* policies can be viewed as a special case of *bounded path* policy, and can be encoded in a similar way. When $m > 1$, the edge expresses a *multipath* invariant (e.g., $D \rightarrow C$ in Figure 4.4), whereas $m = 0$ specifies that two ends of the edge are required to be *isolated* from each other (e.g., $E \rightarrow C$ in Figure 4.4). Furthermore, concatenating edges together can denote *service chaining* type of policies. As shown in Figure 4.4, traffic from node A should traverse a waypoint B before reaching C. A special case worth mentioning is how to express a *load balancing* policy, as this policy typically requires distributing traffic in a certain way among a pool of servers. In our policy model, this can be expressed by assigning m a fractional

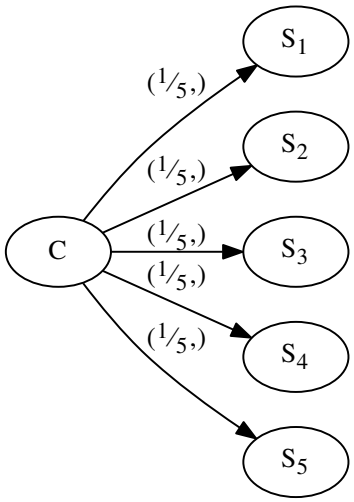


Figure 4.5: *Load balancing policy*

value, in contrast to the integer values it takes in the previous examples. Figure 4.5 denotes a policy that requires traffic from client C distributed evenly among five servers. In this way, *NEAt* uses *policy graphs* to express both qualitative and quantitative reachability constraints on the network.

4.4 Repair Algorithm

In this section, we present *NEAt*'s core algorithm for repairing violations in real-time constrained by a given policy graph. First, we introduce the network model and give an overview of the algorithm. Next, we describe our formulation of the repair problem for basic reachability policies as an integer linear programming (ILP) problem. We then generalize this approach to repair the wider range of policies discussed in 4.3.

Network Model As described in §4.2, upon intercepting an update, *NEAt* constructs a network graph model for each affected EC that captures the configure forwarding behavior for all packets within the EC. This directed *configuration graph* ℓ_c, along with a topology graph T and policy graph $\wp$ serve as inputs to the repair algorithm.

Each node in these graphs represents a host or a networking device, and each edge between a pair of nodes defines reachability between them. The policy graph $\wp$ is a directed graph constructed

from a set of conflict-free policies that represents the expected behavior of the whole network and hence should not be violated at runtime. Each node in policy graph represents a device or a group of equivalent devices, and each edge (u, v) represents an expected path from u to v. Policies' conflict freedom can be guaranteed by tools like PGA [59], which is out of the scope of this study. A topology graph T is an undirected graph that represents the physical topology of the network.

Algorithm Overview When the verification engine discovers a violated EC, the algorithm is executed. Its goal is to repair the detected violations optimally, i.e., with the minimum number of changes to the original configuration. Upon receiving the violated EC c together with its configuration graph ℓ_c, $NEAt$ formulates the problem as an optimization problem: we aim to add or delete the minimum number of edges on ℓ_c so that the modified ℓ_c complies with $\wp_c$. $\wp_c$ is a subgraph of $\wp$ that is relevant to EC c. Note that the added or deleted edges are constrained within the topology graph T. We solve the optimization problem using ILP.

Subsection §4.4.1 describes the repair algorithm for basic reachability policies, and subsection §4.4.2 enhances the basic algorithm to cope with the entire set of policies in §4.3. We complete the section with our repair algorithm for forwarding loops (§4.4.3). Table 4.1 summarizes the key notations used in this section and the next section §4.5.

Table 4.1: Key notations in problem formulation.

Symbol	*Description*
ℓ_c	The configuration graph for equivalence class c.
$\wp$	The policy graph.
T	The topology graph.
(i, j)	The edge from node i to node j.
ρ_{ij}	The paths between node i and node j.
C_i^c	The cluster of node i for equivalence class c.
c_i	The compressed node i for C_i^c.
$E(a)$	The set of all edges in graph a.
$N(E(a))$	Number of all edges in graph a.
$NB_a(i)$	The set of all neighbors of node i in graph a.

4.4.1 Repair Basic Reachability

After receiving a configuration graph ℓ_c that violates the desired policies from the verification engine, the optimizer determines the minimum number of edges that needs to be added or deleted to ensure ℓ_c is consistent with the policy graph $\wp_c$ using Integer Linear Programming (ILP). We start with the basic case where $\wp_c$ contains only reachability constraints.

Our integer program has a set of binary decision variables $x_{i,j,p,q}$ and $x_{i,j}$ where

$$x_{i,j,p,q}, (i,j) \in E_T, (p,q) \in E_{\wp_c} \tag{4.1}$$

$$x_{i,j}, (i,j) \in E_T \tag{4.2}$$

E_T and $E_{\wp_c}$ denote the set of all edges in T and $\wp_c$ respectively. Variable $x_{i,j,p,q}$ defines the mapping between a physical edge and a policy graph edge. It is 1 if a directed edge (i,j) is mapped to policy edge (p,q) for the current EC c, i.e., the flow from p to q will be forwarded through edge (i,j) from i to j. Variable $x_{i,j}$ defines whether or not edge (i,j) is used for forwarding this EC's traffic regardless of which flow uses it. Edge (i,j) in T is selected if any flow (p,q) is forwarded through (i,j) (Equation 4.3). Similarly, for the other direction (j,i), we have Equation 4.4. No physical link can be selected to forward traffic for the same EC on both directions (Equation 4.5) to avoid tight loops.

$$\forall (i,j) \qquad x_{i,j} \geq \sum_{(p,q) \in E_{\wp_c}} \frac{x_{i,j,p,q}}{N(E_\wp)} \tag{4.3}$$

$$\forall (j,i) \qquad x_{j,i} \geq \sum_{(p,q) \in E_{\wp_c}} \frac{x_{j,i,p,q}}{N(E_\wp)} \tag{4.4}$$

$$\forall (j,i) \qquad x_{i,j} + x_{j,i} \leq 1 \tag{4.5}$$

Equations 4.6-4.8 are the flow conservation equations for policy level reachability (p,q).

$\forall (p,q), \forall i \in T$:

$$\begin{cases} \sum_{j \in NB_T(i)} x_{i,j,p,q} = 1 \\ \sum_{j \in NB_T(i)} x_{j,i,p,q} = 0 \end{cases} \quad \text{if } i = p \tag{4.6}$$

$$\begin{cases} \sum_{j \in NB_T(i)} x_{i,j,p,q} = 0 \\ \sum_{j \in NB_T(i)} x_{j,i,p,q} = 1 \end{cases} \quad \text{if } i = q \tag{4.7}$$

$$\begin{cases} \sum_{j \in NB_T(i)} (x_{i,j,p,q} - x_{j,i,p,q}) = 0 \end{cases} \quad \text{otherwise} \tag{4.8}$$

The optimization objective is to minimize the number of changes (additions and deletions) on the original configuration graph ℓ_c.

$$\min \Big(\sum_{(i,j) \notin E_{\ell_c}} x_{i,j} - \sum_{(i,j) \in E_{\ell_c}} x_{i,j} \Big) \tag{4.9}$$

4.4.2 Generalizing the Algorithm

To support generalized reachability policies as discussed in §4.3, we encode several additional constraints into the ILP.

Isolation We introduce a special $DROP$ node. If two nodes are required to be isolated, i.e., the nodes are connected with a $(0,)$ edge in the policy graph, we change the way flow conservation equations are defined. More specifically, Equation 4.7 is changed to Equation 4.10. That is, a flow from p to q should sink at $DROP$ before reaching q.

$$\begin{cases} \sum_{j \in NB_T(i)} x_{i,j,p,q} = 0 \\ \sum_{j \in NB_T(i)} x_{j,i,p,q} = 1 \end{cases} \quad \text{if } i = DROP \tag{4.10}$$

Service Chaining With service chaining, or waypointing, we enhance our flow conservation equations with Equation 4.11. This extends it beyond individual reachability requirements in the policy, and takes into account dependencies between policy edges. The resulting mapping is guaranteed to satisfy chaining of reachability requirements. For instance, if a policy node i is required to reach q through p, because of this equation, i cannot be mapped to the path segment

(p, q). Otherwise, p might be skipped on the path from i to q.

$$\begin{cases} \sum_{j \in NB_T(i)} x_{i,j,p,q} = 0 \\ \sum_{j \in NB_T(i)} x_{j,i,p,q} = 0 \end{cases} \quad \text{if } i \in \wp_c \text{ and } (\exists \rho_{i,p} \text{or } \exists \rho_{q,i}) \tag{4.11}$$

Bounded or Equal Path Length A special case is shortest path policy, where the bounded length is the length of the shortest physical path. If a path length bound n is specified for a policy edge (p, q), then a new constraint is added (Equation 4.12):

$$\sum_{(i,j) \in E_T} (x_{i,j,p,q} + x_{j,i,p,q}) \leq n \tag{4.12}$$

Multipath If at least m link-disjoint paths are required for flow (p, q), then the flow conservation equations 4.6 and 4.7 are updated as Equation 4.13 and 4.14 respectively.

$$\begin{cases} \sum_{j \in NB_T(i)} x_{i,j,p,q} \geq m \\ \sum_{j \in NB_T(i)} x_{j,i,p,q} = 0 \end{cases} \quad \text{if } i = p \tag{4.13}$$

$$\begin{cases} \sum_{j \in NB_T(i)} x_{i,j,p,q} \geq m \\ \sum_{j \in NB_T(i)} x_{j,i,p,q} = 1 \end{cases} \quad \text{if } i = q \tag{4.14}$$

Load Balancing As discussed in §4.3, policy edges within a load balancing policy are denoted with a decimal path count. Correspondingly, in our optimization problem, variables that map physical edges to policy edges are also decimal values between zero and one, instead of binary values. In addition to that change, We introduce a new equation (Equation 4.15) to capture how flow distribution propagates.

$$\prod_{x_{i,j,p,q} \neq 0} x_{i,j,p,q} = m \tag{4.15}$$

For example, consider a physical topology shown in Figure 4.6, where there are two layers of load balancing between client C and servers $S1 - S5$. If the policy in Figure 4.5 is required, the solutions for variables $(x_{i,j})$ are shown in Figure 4.6.

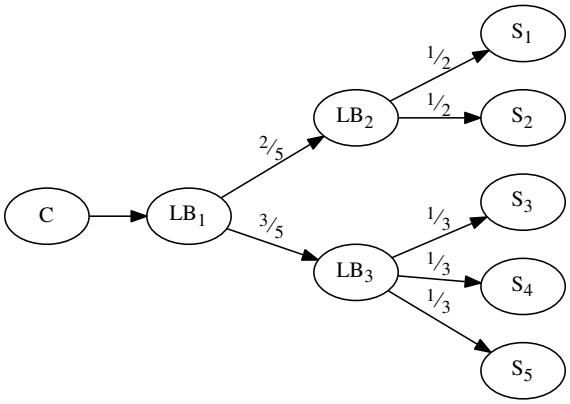


Figure 4.6: *Load Balancing Configuration.*

4.4.3 Repairing Loops

The preceding repair algorithm operates on a *loop-free* configuration graph. As such, we first check for and remove loops from each configuration graph before compressing and repairing violations of any other property type. Our objective for repairing loops is to minimize change to the network, with a preference to affect few equivalence classes as possible, as well as removed the minimal number of rules. Thus, our algorithm will remove a forwarding rule matching packets destined to 10.0.0.1/32 over one for 10.0.0.0/8. Since loops are repaired first, and $NEAt$ will later check reachability properties on each equivalence class, our loop repair algorithm does not need to consider introducing permanent reachability violations by removing rules.

Algorithm 4.1 Loop repair

```
procedure RemoveLoop(ℓ_c, Θ(c))
    # remove edges appearing in multiple loops
    remove {(i,j) | (i,j) ∈ θ_k ∧ (i,j) ∈ θ_m ∀k,m ∈ Θ(c)}
    if N(Θ(c)) = 0 then
        return ℓ_c
    for all θ_i ∈ Θ(c) do
        while N(θ_i) > 0 do
            # remove edges forwarded out the destination
            remove (i,j) if i is destination
        while N(θ_i) > 0 do
            # remove most specific forwarding rule
            remove (i,j) ∈ θ_i with longest prefix
    return ℓ_c
```

Algorithm 4.1 presents our loop repair algorithm. $\Theta(c)$ denotes the set of all loops appearing in a configuration graph ℓ_c and $N(\Theta(c))$ the number of loops in ℓ_c. θ_i is a subgraph of ℓ_c, and $N(\theta_i) = 0$ when the subgraph contains no loops. The algorithm begins by finding and removing all intersecting edges across ℓ_c's loops. For each loop in ℓ_c that is not repaired by removing these edges, next remove an edge (i, j) where i's IP address is the destination, if such an edge exists. While θ_i still has loops, remove an edge in the loop which has the most specific match rule (e.g., longest prefix). Each edge is mapped to a specific forwarding rule at a particular switch when we compute the equivalence classes.

Removal of a forwarding rule is accomplished by replace it with a drop rule, to prevent a coarser match in a separate equivalence class from introducing another loop. For example, if a rule matching destination IP 10.0.0.1/32 is simply deleted from a switch's forwarding table, another rule matching 10.0.0.1/31 on the same switch and forwarding to the same next hop could prevent the loop from being repaired. To conserve switch memory during in response repairs, *NEAt* checks all coarser drop rules to determine if multiple rules can be aggregated together.

4.5 Optimizations

While conceptually straightforward, the mapping algorithm in section 4.4 does not scale to a large network. In the optimization problem formulation, the number of variables for one EC is approximately the product of the number of edges in the physical topology and the number of edges of the policy graph, which can easily exceed 100k. In this section, we present two techniques that dramatically optimize the repair speed.

4.5.1 Topology Limitation

This technique aims to "slice" away irrelevant or redundant part of the network, and thus shrink the size of the optimization problem. After getting a forwarding graph that violating some policy, before forming the optimization mapping problem, we first remove disconnected component on the physical topology. Next, we localize the potential affected area on the topology. Fortunately, most modern networks are designed in a hierarchical structure. Examples include data centers arranged in a fattree topology, and enterprise networks divided into multiple sites joint by a backbone

network. Such a structure implies certain communication pattern: communication within a subtree should stay local, for example, and communication between subtrees normally doesn't traverse other subtrees, i.e., go through a valley. Based on this, in our linear programming problem, typically only a subset of the topology is considered mappable to a policy edge. Results in section 4.7 shows the effectiveness of this technique.

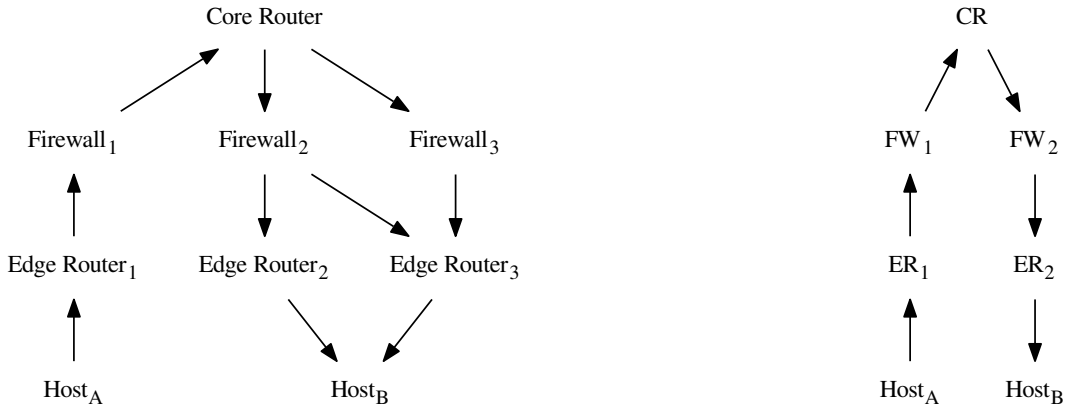
4.5.2 Graph Compression

Besides hierarchical structures, most large networks are designed in patterns that enforce symmetry to some extent [57] for load balancing or resilience reasons. For example, in a data center fattree topology, devices on the same layer (access, aggregate, core) are symmetrically connected to multiple devices on the neighboring layers. We exploit such regularities to compress the graphs that the repair algorithm operates on. The key to the compression is that the compressed graphs must be equivalent to the original graphs with respect to the policies of interest. To this end, we leverage the *graph pattern preserving* algorithm [25] as the major building block of $NEAt$'s compressor (Figure 4.1). The algorithm compresses a labeled directed graph according to the *bisimulation relation*.

Bisimulation Relation [24] We denote $G = (V, E, L)$ as a labeled directed graph. V represents a set of node and $(u, v) \in E$ represents a directed edge from node u to node v. $L(u) \in \Gamma$ represents the label of node u, where Γ is the set of labels that applied to V. In the networked system context, the labels may represent a set of similar functional networking nodes, e.g. hosts, firewalls, load balancers. For example, in Figure 4.7(a), we label the network nodes as *Firewall*, *Edge Router* and *Core Router* and we label the two hosts as $HostA$ and $HostB$.

A *bisimulation relation* on a graph $G = (V, E, L)$ is a binary relation $BR \subseteq V \times V$ such that for each $(u, v) \in BR$, (1) $L(u) = L(v)$; (2) for each $(u, u') \in E$, there exists an edge $(v, v') \in E$ such that $(u', v') \in BR$; (3) for each $(v, v') \in E$, there exists an edge $(u, u') \in E$ such that $(u', v') \in BR$. One can verify that in Figure 4.7(a) $Firewall_2$ and $Firewall_3$ are *bisimular* to each other, while $Firewall_1$ is not *bisimular* to any other firewall. Because $HostB$ is solely in a *bisimular* cluster, and hence $EdgeRouter_1$ and $EdgeRouter_2$ are *bisimular* as they only has one child $HostB$. As $Firewall_2$ and $Firewall_3$ have the children that are *bisimular*, they are also *bisimular* to each other.

While $Firewall_1$'s child is *Core Router*, which has a different label than *Edge Router*, $Firewall_1$ is not *bisimular* to anyone.



(a) Original graph

(b) Compressed graph

Figure 4.7: *Example of compression*

Bisimulation Based Compression Algorithm 4.2 presents the compression algorithm on the given graphs ℓ_c, $\wp_c$ and T. We compute *bisimulation relation* on ℓ_c using the algorithms presented in [24] and then compress the graphs based on the bisimularity. However, unlike ℓ_c and $\wp_c$, T is not a directed graph, and thus the original algorithm is not applicable. To compute T^{cp}, we first compress the parts in T that overlap with ℓ_c according to the undirected version of ℓ_c^{cp}. Then we draw edges between the non-overlapping parts and the compressed parts with their original edges in T. The time complexity of the compression algorithm is $O(|E|log|V|)$. Figure 4.7(b) shows the compression result on graph ℓ_c. $Firewall_2$ and $Firewal_3$ are *bisimular* and are compressed to a new clustering named FW_2. $Firewall_1$ stays by itself as FW_1. $\wp_c$ is compressed in a similar way.

Algorithm 4.2 Graph pattern preserving compression

procedure CLUSTERING(ℓ_c, $\wp_c$, T)
compute the maximum bisimulation relation BR of ℓ_c
compute the clusters $clusters = V/BR$
collapse the nodes in the each $cluster \in clusters$
compute compressed $\wp_c$, ℓ_c^{cp}, T^{cp}
return $\wp_c$, ℓ_c^{cp}, T^{cp}

We evaluate the compression algorithm on a simulated fattree topology and a large enterprise network. We denotes the compression rate r_c as the ratio of the number of the remaining nodes in ℓ_c^{cp} to the number of the nodes in the original graph ℓ_c. Table 4.2 shows the compression results. From the result we can conclude that the compression algorithm could result in a much smaller amount of nodes for large-scale networks.

Table 4.2: Compression results.

Topology	$1 - r_c$
Fattree (6750 hosts, 1125 switches)	99.38%
Enterprise (236 routers)	88.98%

Incremental Compression Further leveraging the incremental compression algorithm from [25], we incrementally maintain the compressed configuration graphs. In response to changes to the original graphs, the incremental algorithm computes the new compressed graph using the changes and the compressed graph as input, independent of the original graph. That is there is no need to decompress the graph to propagate the changes.

Repair Compressed Graphs With the compression module in place, when a violation is detected, the graphs are compressed first, then passed to the optimizer. Note that one compressed topology graph edge may represent a collection of original topology graph edges. This works fine with single-path reachability type of policies, such as reachability, isolation, service chaining. However, it will break equation 4.13 and 4.14 for link-disjoint multipath policy. Our solution is to label predecessors of a multipath policy destination (E.g., q for policy edge (p, q)) node differently, such that they are not compressed into one cluster. In addition, the compressed topology graph is modeled as a weight graph, where the weight on each edge is the number of original edges the compressed edge represents. Multipath policy constraint Equation 4.13 is modified accordingly as Equation 4.16, where as Equation 4.14 remains the same because there's never multiple edges pointing to the destination node q.

$$\begin{cases} \sum_{j \in NB_{T^{cp}}(i)} (x_{i,j,p,q} * weight_{i,j}) >= m & \\ \sum_{j \in NB_{T^{cp}}(i)} x_{j,i,p,q} = 0 & \text{if } i = p \end{cases} \tag{4.16}$$

Map Back The last step is to map the result back to the original graphs. The optimization result is a set of changes (added or deleted edges) on the compressed graphs. To map back to the original configuration graph, a changed edge (c_i, c_j) could become a set of changed edges between the cluster C_i^c and cluster C_j^c. If an edge (c_i, c_j) is supposed to be added to the compressed configuration graph, then on the original configuration graph, for every node i in the source cluster C_i^c, there should be an edge added from i to *one* of its neighbor node j that is in the target cluster C_j^c. By the definition of bisimulation, such a node j always exists. On the other hand, if an edge (c_i, c_j) should be removed from the compressed configuration graph, then *all* the edges between the two clusters should be removed on the original graph.

Afterwards, those computed changes will be translated into forwarding instructions, and sent to the network devices.

Policy Perseverance At the end, we prove that the compression algorithm preserves the equivalence between the compressed graph G_c and the original graph G with respect to the scope of policies in section 4.3. The equivalence is proved in [25] for graph pattern queries. A graph pattern query is effectively asserting single-path type reachability and bounded path length. So here we only need to prove that the conclusion also holds for multipath policies.

Theorem 4.1. *(Multipath Equivalence): A multipath policy for a flow (p, q) holds on G* iff *the policy also holds for (p, q) on G_c.*

Proof. Consider a multipath policy that requires at least m path for flow (p, q). Trivially, if a flow (p, q) satisfies the policy on G, the policy also holds for (p, q) on G_c, and flow conservation equations (Equation 4.16, 4.14 and 4.7) are satisfied. So it's effectively to prove that the policy must hold when Equation 4.16 4.14 and 4.7 are satisfied.

Let $path_1^c$, $pach_2^c$, ... $path_n^c$ $(n \leq m)$ be the set of paths from p to q on G_c that collectively satisfy Equation 4.16, 4.14 and 4.7. If n equals m, then there are m link-disjoint paths on G_c between p and q, and thus there are m link-disjoint paths on G, i.e., the policy is satisfied.

If n is less than m, then there must be at least a path on G_c, whose starting edge's weight is more than one. Let those paths be $path_{m0}^c$,...,$path_{mj}^c$, whose starting weights are k_0, ..., k_j respectively. Consider path $path_{m0}^c$ first. This means its starting edge is pointing from p to a cluster C_{next} which

contains at least k_0 nodes which are also p's successors. Because the predecessors of q are labeled differently, each of them is a separate cluster. By definition of bisimulation relation, two nodes are bisimular (and thus can be clustered together) only if their children's label set are the same. Via back propagation, and constraint by Equation 4.7, there must be at least k_0 disjoint paths on G from p's successors in C_{next} to q's predecessors. Similarly, iterate through all the paths from p to q on G_c, there are at least m paths from p to q on G. □

4.6 Implementation

We implemented a prototype of *NEAt* in Python. *NEAt* requires no modifications to the controller or switches. The verification engine is based on our prior work [43] and we use the Gurobi Optimizer [6] within our optimization engine to solve the ILP.

NEAt's pass-through mode is implemented as a proxy between the controller and switches, listening for flow modification messages. The interactive mode is implemented as an XML-RPC API, allowing it to be compatible with applications written in any language or for any controller. In particular, *NEAt* exposes a `check()` function that accepts a set of OpenFlow flow modification messages to check against the network policy. *NEAt* updates the network model with the proposed changes, verifies the model, and searches for a set of repairs if any violations are found. The application can choose to receive the repairs as a set of OpenFlow flow modification messages or as a set of edge tuples. For example, a load balancer application may wish to receive a repairs as a set of tuples (e.g., *[(s2, h1)]*) to easily re-assign a client to a particular server replica, rather than parsing an OpenFlow message from *NEAt*.

4.7 Evaluation

In this section, we examine the performance of repairs in *NEAt*, as well as the end-to-end latency experienced by applications.

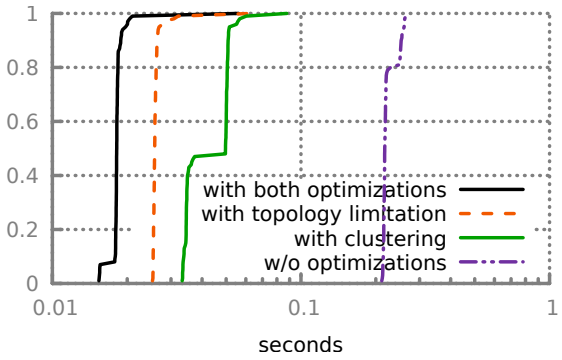
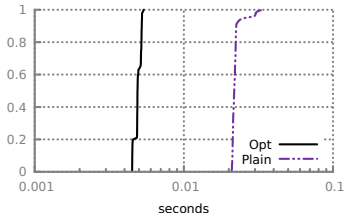
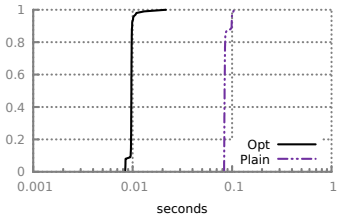


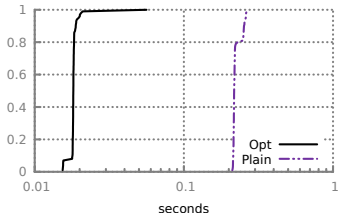
Figure 4.8: *Effect of optimizations*



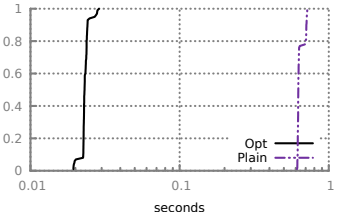
(a) 54 hosts, 45 switches



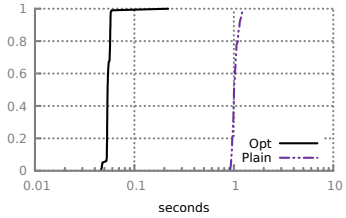
(b) 128 hosts, 80 switches



(c) 250 hosts, 125 switches



(d) 432 hosts, 180 switches



(e) 686 hosts, 245 switches

Figure 4.9: *Repair time comparison under random removals of exactly matching rules*

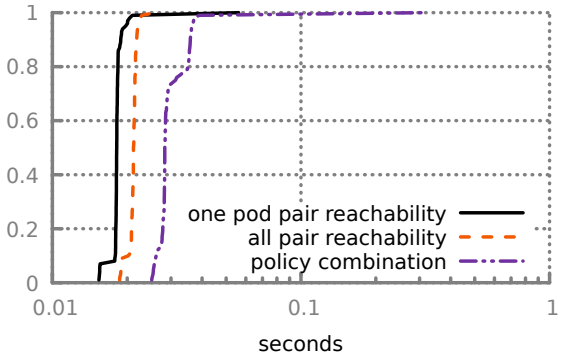


Figure 4.10: *Different policies.*

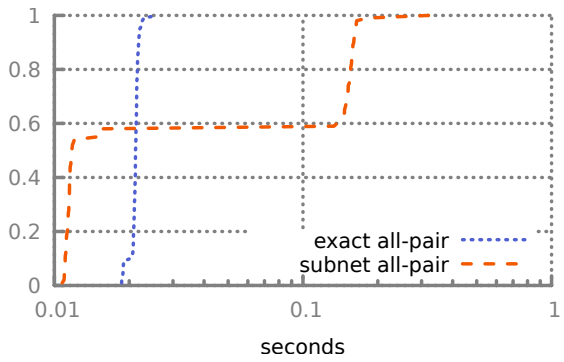


Figure 4.11: *Exact matching rules vs. overlapping rules*

4.7.1 Repair Performance

To evaluate the feasibility and scalability of *NEAt*'s repair process, we synthesized a set of fattree topologies with various sizes, and used *NEAt* to maintain a variety of network-wide policies, including reachability, segmentation, bounded path length and multipath policies. More specifically, on each topology, under random removals of rules, we measured the repair time for each removal that caused a violation.

4.7.2 Exact matching rules

We first focus on flow-based traffic management applications, which are widely used in SDN [18, 32, 34, 35, 37]. Any forwarding rule produced by such applications at a switch matches at most one flow. In our terms, each rule only affects at most one EC.

For each fattree topology, we randomly selected a pair of pods. Suppose the desired policy is that any host in one of the pods should be able to reach every other host in both selected pods, which we will refer to as a *pod-pair reachability* policy. With random removals of rules, for those removals resulting in violations to *pod-pair reachability*, the optimization engine is triggered to perform the repair. For testing purposes, we re-verify the policy after each repair, and the check passed for all cases.

On a fattree topology with 250 hosts and 125 switches, we measured the time taken to repair *pod-pair reachability* policy by four mechanisms: (1) plain mapping, (2) mapping with topology limitation, (3) mapping with graph compression, and (4) mapping with both compression and topology

limitation. Figure 4.8 compares the CDFs of the repair time for these four repair mechanisms: We can see the combination of graph compression and topology limitation (left most curve) brings approximately one order of magnitude speed-up over plain mapping (right most curve). Figure 4.9 (a-e) shows the amount of speed-up goes up as the network size scales. Even on a network with 686 hosts and 245 switches, the repair time is bounded under 0.1 second for the majority case, close to 1/20 of the repair time by plain mapping.

We next explored how *NEAt* handles a larger set of policies and a combination of different types of policies. We first assumed the desired policy being every pair of hosts should be able to reach each other , which we will refer to as an *all-pair reachability* policy. Again, on a fattree topology with 250 hosts and 125 switches, the repair time under random rule removals against this *all-pair reachability* policy was measured, as shown in Figure 4.10. The policy size is increased by approximately 10 times compared with *pod-pair reachability* policy, but the repair time only increases slightly.

To test a even more complex setting, next we randomly selected three pods in the fattree. Between the first two pods, hosts should be isolated from each other (*segmentation*), and between the first and third selected pods, hosts are connected by at least two path (*multipath*). For host pairs that do not fall into the previous two conditions, they are supposed to be able to reach each other (*all-pair reachability*). Both *multipath* and *all-pair reachability* are combined with a bounded path length policy, to avoid flows between pods "go through a valley". Note that unlike the previous pure single-path reachability policy, where repairs are all edge additions, in this case, a repair is sometimes a mix of edge additions and deletions. What's more, to satisfy multipath requirement, more additions are necessary. Due to this complexity, the repair time is increased, but still on the same order of magnitude of reachability policy cases, as shown in Figure 4.10. As verified by the re-checks, changes for fixing different types of policies keep other policy intact.

4.7.3 Overlapping rules

For networks that use wild-carded rules or longest prefix matching, the assumption in the previous subsection does not hold. One rule may affect multiple ECs, and thus potentially trigger repairs on multiple graphs. Fortunately, there is a trend to move such overlapping rules to network

edge or even hosts [8, 20, 48], leaving the core with exactly matching rules. In order to study how *NEAt* performs under this less preferable but less common scenario, we assign IP addresses within the same prefix subnet to hosts within the same pod on the fattree topologies. We then aggregated rules on the switches as much as possible. For example, each core switch has only k forwarding rules, where k is the number of pods, and each rule matches on one pod's prefix. Similar to the previous experiments, we used *NEAt* to guarantee an *all-pair reachability* policy, and our engine discovered repairs for all violations. Figure 4.11 compares the CDFs of the repair time for overlapping rules and exact matching rules on a 250-host-125-switch fattree topology. The repair took longer compared to applications with exact match rules because of the increased number of affected ECs. With our graph compression and topology limitation techniques, optimization is able to finish under 0.4 seconds in the worst case.

4.7.4 End-to-End Delay

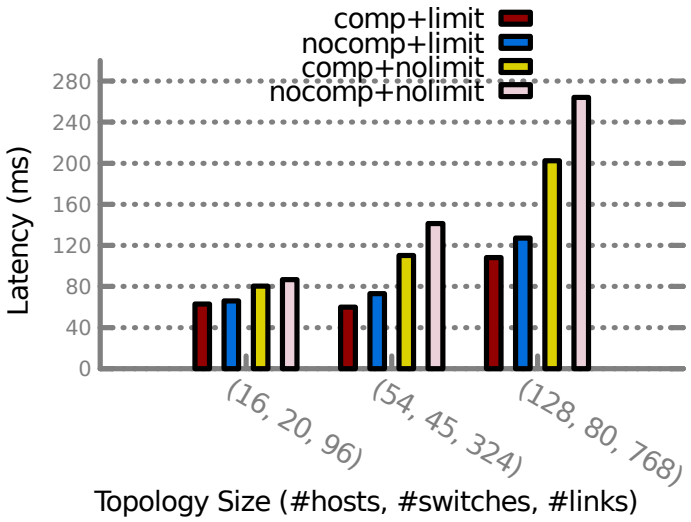


Figure 4.12: *Application-perceived latency of NEAt, on various fattree topologies, showing performance for a reachability policy with/without graph compression and topology limitation*

Next, we examine the application-level delay introduced by *NEAt* when using its interactive mode. We test *NEAt* on various-sized fattree topologies using Mininet [7] and the Pox controller [13]. A learning switch application and load balancer application run on top of Pox. The load balancer balances flows between the two replicas in a round-robin fashion, and we modify it to

leverage *NEAt*'s API to check the assignment of clients to replicas. If *NEAt* suggests a repair, the application updates its client-to-replica mapping with one suggested by *NEAt*. While the learning switch remains unmodified and unaware of *NEAt*, its updates are transparently checked by *NEAt*. This setup demonstrates the ability of *NEAt* to interact with the controller and applications simultaneously through its two interactive modes.

The load balancer application runs on an edge switch in the fattree topology, with clients and server replicas placed in different pods. To trigger an update, a client pings the virtual IP of the load balancer. When the appropriate event handler in the load balancer is execute, it invokes *NEAt*'s `check()` function. We measure the total latency introduced by *NEAt* as the time to invoke the `check()` function and apply it to the application's state. This includes the time to verify an update (i.e, calculate equivalence classes affected by the update, compute their configuration graphs, and verify them) and repair violations in any of the affected equivalence classes.

For each topology size, we examine the total latency for a reachability policy, with and without our compression and topology limitation optimizations. Figure 4.12 shows the total delay experienced by the load balancer. Topology limitation has the largest speed-up of our optimizations, but when used in combination with compression of the topology and configuration graphs, *NEAt* can verify and repair an update in under 120ms.

4.7.5 Enterprise Network Trace Study

Finally, we examine traces from a large enterprise network, to examine *NEAt*'s performance on real forwarding graphs. We examine two dumps of the data plane from 2014 and 2017. These datasets containing more than one million forwarding rules across more than 200 forwarding devices. The 2014 dataset contains 27k equivalence classes, while the 2017 trace contains 285k.

4.7.6 Bugs found in a real network

For each dataset, we construct loop and reachability policies and check for violations. In the 2014 dataset, *NEAt* finds nine different loops. In the 2017 dataset, *NEAt* finds 19. We examine the forwarding table and find several of these are caused by default routes with prefix 0.0.0.0/0. Only equivalence classes with more specific rules on the device are free of loops in these cases.

Another cause we discover is load balancing – a device can forward packets out one of two ports, one of which will result in a path containing a loop.

4.7.7 Simulation

We next use the 2017 dataset to evaluate *NEAt*'s scale and performance on a data plane with a realistic number of equivalence classes. First, we verify and repair any loops in the 285k equivalence classes contained in the dataset. We then construct artificial updates, choosing a destination IP address and prefix length with the same probability as they appear in the dataset's forwarding rules. An update can either add a rule, delete a rule, or introduce a loop. Loops are chosen from the list of those that were discovered and repaired at the start of the simulation. An update has a 10% chance of introducing one of these loops for a particular update, which may introduce loops in multiple ECs. We simulate 100 updates, affecting an average of eight ECs per update.

We apply the set of random updates to different combinations of policies, including loop-freedom, reachability, and our compression and topology limitation optimizations. Since the compression and topology limitation optimizations only apply to the reachability policy, we do not test loop freedom in combination with compression or topology limitation. Figure 4.13 shows a CDF of the total update time, including verification and repairs (when repairs are necessary). Of the 100 updates, 20 loops violations needed repair, as well as 24 reachability violations. Median and 98th percentile update times were 10ms and 1300ms, respectively, for a reachability policy with compression and topology limitation enabled. For a loop freedom property, median and 98th percentile update times were 35ms and 730ms, respectively. Combining these two policies, without compression or topology limitation optimizations, resulted in median and 98th percentile times of 36ms and 193 seconds. Adding our two optimizations reduced these times to 36ms and six seconds.

4.8 Summary

In this chapter, we presented *NEAt*, a system that provides network administrators with a network analogue of a smartphone's autocorrect. As a transparent layer, *NEAt* repairs, in real-time, updates from an SDN controller that violate generic policies such as reachability, service-chaining, and segmentation. *NEAt* repairs the updates by adding or removing a minimal number of rules in

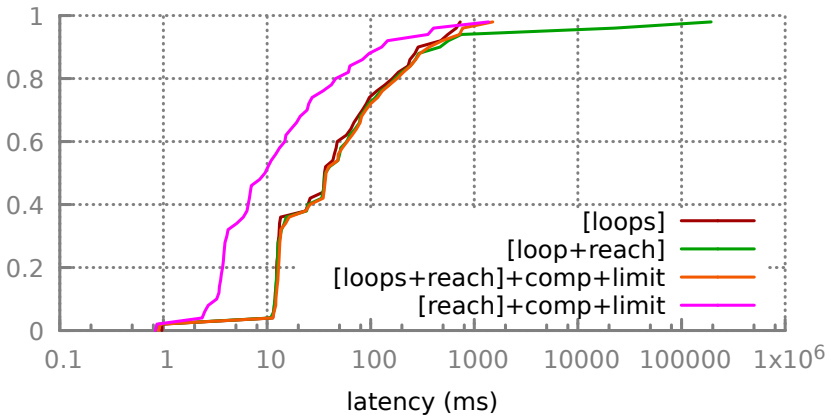


Figure 4.13: *Total update time comparison for different combinations of policies and optimizations, on a model of a real-world data plane trace*

order to comply with the policy by casting the problem as an optimization problem. Preliminary experiments on large fattree topologies show our optimization problem-based formulation can discover repairs in under one second for applications with non-overlapping rules, and two seconds for applications issuing rules with more complex dependencies. Applying *NEAt* to a large enterprise network uncovered and repaired 28 loops across two data plane snapshots.

Chapter 5

Related Work

In this chapter, we review existing approaches to network diagnosis and enforcing correctness into networks.

5.1 Network Debugging

Previous work on debugging general networks and SDNs focuses on detecting network anomalies [17, 47], checking OpenFlow applications [22], ensuring data-plane consistency [50, 63], and allowing multiple applications to run side-by-side in a non-conflicting manner [52, 58, 65]. However, unlike our work, none of the prior solutions provides real-time verification of network-wide invariants as the network experiences dynamic changes.

Checking OpenFlow applications: Several tools have been proposed to find bugs in OpenFlow applications and to allow multiple applications run on the same physical network in a non-conflicting manner. NICE [22] performs symbolic execution of OpenFlow applications and applies model checking to explore the state space of an entire OpenFlow network. Unlike *VeriFlow*, NICE is a proactive approach that tries to figure out invalid system states by using a simplified OpenFlow switch model. It is not designed to check network properties in real time. FlowVisor [65] allows multiple OpenFlow applications to run side-by-side on the same physical infrastructure without affecting each others' actions or performance. Unlike *VeriFlow*, FlowVisor does not verify the rules that applications send to the switches, and does not look for violations of key network invariants.

In [52], the authors presented two algorithms to detect conflicting rules in a virtualized OpenFlow network. In another work [58], Porras et al. extended the NOX OpenFlow controller with a live rule conflict detection engine called FortNOX. Unlike *VeriFlow*, both of these works only detect conflicting rules, and do not verify the forwarding behavior of the affected packets. Therefore, *VeriFlow* is capable of providing more useful information compared to these previous works.

Checking network invariants: The router configuration checker (rcc) [26] checks configuration files to detect faults that may cause undesired behavior in the network. However, rcc cannot detect faults that only manifest themselves in the data plane (e.g., bugs in router software and inconsistencies between the control plane and the data plane; see [47] for examples).

Anteater [47] uses data-plane information of a network, and checks for violations of key network invariants (absence of routing loops and black holes). Anteater converts the data-plane information into boolean expressions, translates network invariants into instances of boolean satisfiability (SAT) problems, and checks the resultant SAT formulas using a SAT solver. Although Anteater can detect violations of network invariants, it is static in nature, and does not scale well to dynamic changes in the network (taking up to hundreds of seconds to check a single invariant). Header Space Analysis [40] is a system with goals similar to Anteater, and is also not real time.

Concurrent with our work, NetPlumber [39] is a tool based on Header Space Analysis (HSA) that is capable of checking network policies in real time. NetPlumber uses HSA in an incremental manner to ensure real-time response. Unlike *VeriFlow*, which allows users to write their own custom query procedures, NetPlumber provides a policy language for network operators to specify network policies that need to be checked.

ConfigChecker [17] and FlowChecker [16] convert network rules (configuration and forwarding rules respectively) into boolean expressions in order to check network invariants. They use Binary Decision Diagram (BDD) to model the network state, and run queries using Computation Tree Logic (CTL). *VeriFlow* uses graph search techniques to verify network-wide invariants, and handles dynamic changes in real time. Moreover, unlike previous solutions, *VeriFlow* can *prevent* problems from hitting the forwarding plane, whereas FlowChecker find problems after they occur and (potentially) cause damage. ConfigChecker, like rcc, cannot detect problems that only affect the data plane.

An early version of *VeriFlow* was presented in [42] but only supported checking for a single header field with a basic reachability invariant, had comparatively high overhead, and had a limited evaluation.

5.2 Ensuring Correctness

Orthogonal to detecting violations, researchers have also investigated techniques to rigorously preserve network properties. On one hand, from time dimension, there's effort on how to synthesize a feasible network update sequence in order to maintain desirable properties consistently. On the other hand, there's also related work on how to build a *correct* network in the first place, with better programming language abstractions or synthesis tools that construct network states from scratch.

Ensuring data-plane consistency: Another train of inquiry [34, 38, 45, 63] focuses on how to synthesizes a correct update plan to avoid inconsistencies in data-plane, which may cause undetected transient faults in the network. However, their solutions are too expensive to achieve real-time performance with heavy flow table storage usage or long updates buffering time. In addition, the existing approaches are not designed to be flexible enough to verify generic network invariants. Reitblatt et al. [62] also proposed a language based on regular expressions for synthesizing fault-tolerant network programs, but the operations have to be performed offline.

Other researchers have also noticed the problem of inconsistent view between SDN-controller and the network states. Peresini et al. [56] proposes a multi-commit transactional semantic at the controller for ensuring consistent packet processing. Heller et al. [31] presents a big picture of cross-layer diagnostic framework for systematic troubleshooting in SDNs, and rigorous network-wide verification which we have explored, is an essential component towards that goal.

Among the related approaches mentioned in §3.1, four warrant further discussion. Most closely related to our work is Dionysus [36], a dependency-graph based approach that achieves a goal similar to ours. As discussed in §3.1, our approach has the ability to support 1) flexible properties with high efficiency without the need to implement new algorithms, and 2) applications with wildcarded rules. [49] also plans updates in advance, but using model checking. It, however, does not account for the unpredictable time switches take to perform updates. In our implementation, CU [63] and VeriFlow [43] are chosen as the fallback mechanism and verification engine. Nevertheless, they are replaceable components of the design. For instance, when congestion freedom is the property of interest, we can replace CU with SWAN [34].

SDN programming languages: Many programming languages have been proposed to provide abstractions to program SDNs, e.g., Frenetic [27], Pyretic [61] and Maple [69]. These allow programmers to compose complex rules without the need to manually resolve conflicts between rules. However, these languages face limitations implementing general policies that deliver higher-level intent, such as expressing middleware functionality or QoS constraints.

SDN synthesis platforms: Network state can also be synthesized from a set of pre-specified correctness conditions. NetGen [64], for example, takes as input a specification using regular expressions to define paths changes and a set of ECs to modify. It uses an SMT solver to find the minimal number of changes. However, similar to Merlin [67] and FatTire [62], this tool is designed to be used as compiler, with performance that is too slow for real-time applications (i.e., minute-scale synthesis). While using NetGen in place of our ILP is possible, doing so would additionally require translating each update into an equivalent NetGen specification. Similarly, Marham [33] proposes a framework for automated repair, but with slower performance — on the order of several seconds for topologies 10s of nodes and links. Margrave [53] analyzes changes to access control policy changes, highlighting to an operator the effect it has on the policy, without suggesting repairs to violations.

SDN policy languages: In *NEAt* project, we developed a way to express network policies as a directed graph to faciliate network automation. The general direction and approach are also adopted by PGA [59]. Similar to *NEAt*, PGA also describes policies using a graph structure. But the types of policies that can be expressed using PGA is limited to reachability and waypointing. In addition to that, *NEAt*'s policy representation can model quantitative reachability constraints, such as bounded path length, resilience and load balancing. However, PGA is proposed as a way not only to express policies, but also to compose different policies together, which is out of the scope and orthogonal to the goal of *NEAt*. For policies that covered by PGA's policy language, it would be desirable to use PGA to compose them into a conflict free graph, and input that graph to *NEAt*. For other types of policy, it would be interesting to explore ways to resolve conflicts, possibly following PGA's approach.

Chapter 6

Conclusions and Future Work

6.1 Contributions

This dissertation has demonstrated that it is practical and efficient to leverage user instructions as guidance to incrementally build/maintain a network state, where desirable properties are automatically preserved all the time. Our vision behind is that to ensure correctness of complicated systems like modern networks, we should combine the computational power of computers and intelligent guidance of humans. With the help of the verification foundation, we are able to know when and what goes wrong in our network in real time. Then we are able to instill correctness into the network from both *time* and *space* dimensions. First, via repeated verifications and guided by dynamic network instructions, we are able to synthesize network update ordering and timing that migrates network between configurations efficiently, making sure no security holes exposed at any intermediate state. One step further, based on bugs detected by the verification engine, our system automatically corrects problematic network instructions. We plan to develop a network data-plane management software that can work with both general networks and SDNs. We will utilize the techniques developed in this thesis to build the different components of this software suite.

6.2 Discussion and Future Work

Deciding when to check: We may not know when an invariant violation is a true problem rather than an intermediate state during which the violation is considered acceptable by the operator. For example, in an SDN, applications can install rules into a set of switches to build an end-to-end path from a source host to a destination host. However, as $VeriFlow$ is unaware of application semantics, it may not be able to determine these rule set boundaries. This may cause our verification engine to report the presence of temporary black holes while processing a set of rules

one by one. One possible solution is for the SDN application to indicate when to check. Moreover, our verification technique may be used with consistent update mechanisms [50, 63], where there are well-defined stages during which the network state is consistent and can be checked.

Handling packet transformations: A transformation rule such as Network Address Translation has two parts – the match part determines the set of packets that will undergo the transformation, and the transformation part represents the set of packets into which the matched packets will get transformed. We can handle this case by generating additional equivalence classes and their corresponding forwarding graphs, to address the changes in packet header due to the transformations. Although we have put substantial effort in optimizing our verification engine in our *CCG* work, including the way it handles packet transformation, it still requires one extra trie traversal every time a packet transformation happens. This works fine if the number of packet transformations occurs only a few times per packet, for instance, VLAN tagging and untagging. However, in the worst case, transformations may happen at every hop (e.g., in an MPLS network), we leave a full design and implementation to future work.

Multiple controllers: Our system assumes it has a complete view of the network to be checked. In a multi-controller scenario, obtaining this view in real time would be difficult. Checking and repairing network-wide invariants in real time with multiple controllers is a challenging problem for the future.

Synthesis Parallelism *CCG* synthesizes network updates with only heuristically maximized parallelism, and in the cases where required properties are not *segment independent*, relies on heavier weight fallback mechanisms to guarantee consistency. When two or more updates have circular dependencies with respect to the consistency properties, fallback will be triggered. One safe way of using *CCG* is to provide it with a strong fallback plug-in, e.g., CU [63]. Any weaker properties will be automatically ensured by *CCG*, with fallback triggered (rare in practice) only for a subset of updates and when necessary. In fact, one can use *CCG* even when fallback is always on. In this case, *CCG* will be faster most of the time, as discussed in §3.4.3.

Utilizing hardware primitives: It would be interesting to investigate how to utilize possible primitives in network hardware to facilitate verification and synthesis, given the progress in

programmable hardware, for instance, P4 [19].

Deployment Experience Although we have deployed our system on real and virtual switches in lab and simulated environments, and tested it with real enterprise network traces, we have no experience in applying it in production networks, and there are unanswered questions. For instance, how general and important are the network properties we identified? How much impact physical network changes impose on data plane state? Such experience will be crucial to understand our limitations and to identify new research directions.

Appendix A

Completeness of the network model

Let us demonstrate that our uncertainty-aware model can accurately capture the view of the network from packets' perspective. We first define the situation when the view of a packet is consistent with the model.

Definition A.1. *A packet P's view of the network is* **consistent** *with the uncertainty-aware model, if at any time point during its traversal of the network, the data plane state that the packet encounters is in the model at that time point. More specifically, at time t, to P if a link l*

- *is reachable, l is in the graph model for P at t;*
- *otherwise, l is definitely not certain in the graph at t.*

Theorem A.1. *Assuming that no physical failures change the data plane, any packet's view of the network is consistent with the uncertainty-aware model.*

Proof. Without loss of generality, assume that the maximum duration of a packet in the network is δ, which is set as the amount of delay added to confirmations. Consider a packet P that enters the network at time t_1 and leaves at t_2 ($t_2 - t_1 \leq \delta$). Assume that P traverses the network in n hops, and when $n = 0$, P enters the network. Clearly the theorem holds for $n = 0$. Consider hop $k, (k \geq 0$ and $k \leq n)$. By induction, at previous hop $(k-1)$, assume that P's view is consistent with the model.

If P encounters a forwarding link at hop k, then there exist two cases. In case 1, the corresponding forwarding rule is intentionally inserted by the controller, and by the time P reaches hop k, the rule is installed. In case 2, the rule is about to be removed, but the action is not done until P has been handled by the rule. Let t_i denote the time of issuing the related command (to add or remove the rule), and t_c the time it is confirmed at the controller as. In case 1, since P reaches the link, $t_i < t$. In the model, that link is modeled as either certain ($t_c \leq t$) or uncertain ($t_c > t$). In case 2, because the link is reachable to P at t, in P's lifetime $[t_1, t_2]$, P's view of the network

state contains that link. t_c cannot be earlier than t, because if it were, P could not reach the link. Because of the delayed confirmation mechanism, if an update u causes the removal of the link, the status of the rule remains as uncertain for an extra δ time in the model until $t_c + \delta > t_2$, which is consistent with P's view. In particular, if $t_i \geq t$, then the link is included as certain in the model until the update is issued (t_i).

If P reaches a location where no forwarding rule is available, there are also two cases. In case 1, some forwarding rules have been issued to handle P at this location, but they have not been applied yet. In case 2, there had been available rules, but they were removed before t. In case 1, t_c is definitely later than t. If it weren't, the rule would be there by t. If $t_i < t$, at t, the forwarding rule is only modeled as uncertain. If $t_i \geq t$, at t, the model does not contain that rule. In case 2, the removal of the rules is issued before t. In the interval $[t_i, t_c + \delta]$, any rule R is modeled as uncertain, and after the interval, R is removed from the model, after P leaves the network. Hence, the model is consistent with the view of P during its lifetime in the network. □

References

[1] http://www.infosecurity-magazine.com/opinions/to-err-is-human-to-automate-divine/.

[2] http://www.verizonenterprise.com/verizon-insights-lab/dbir/2016/.

[3] Beacon OpenFlow Controller. https://openflow.stanford.edu/display/Beacon/Home.

[4] Ccg tech report. http://web.engr.illinois.edu/~wzhou10/gcc_tr.pdf.

[5] Floodlight Open SDN Controller. http://floodlight.openflowhub.org.

[6] Gurobi optimization. http://www.gurobi.com/.

[7] Mininet: Rapid prototyping for software defined networks. http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet.

[8] Network virtualization for cloud data centers. http://tinyurl.com/c9jbkuu.

[9] NuSMV: A new symbolic model checker. http://nusmv.fbk.eu.

[10] Open vSwitch. http://openvswitch.org.

[11] OpenFlow switch specification. http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf.

[12] Rocketfuel: An ISP topology mapping engine. http://www.cs.washington.edu/research/networking/rocketfuel.

[13] The pox controller. https://github.com/noxrepo/pox.

[14] University of illinois ocean testbed. http://ocean.cs.illinois.edu/.

[15] University of Oregon Route Views Project. http://www.routeviews.org.

[16] E. Al-Shaer and S. Al-Haj. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *SafeConfig*, 2010.

[17] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. ElBadawi. Network configuration in a box: Towards end-to-end verification of network reachability and security. In *ICNP*, 2009.

[18] T. Benson, A. Anand, A. Akella, and M. Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh Conference on Emerging Networking Experiments and Technologiesperiments and Technologiesnference on emerging Networking EXperiments and Technologies (CoNEXT)*, page 8. ACM, 2011.

[19] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *ACM Computer Communication Review*, 2014.

[20] B.Raghavan, M.Casado, T.Koponen, S.Ratnasamy, and a. S. S. A.Ghodsi. Software-defined Internet architecture: Decoupling architecture from infrastructure. In *HotNets*, 2012.

[21] Z. Cai, A. L. Cox, and T. S. E. Ng. Maestro: A system for scalable openflow control. http://www.cs.rice.edu/~eugeneng/papers/TR10-11.pdf.

[22] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE way to test OpenFlow applications. In *NSDI*, 2012.

[23] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling flow management for high-performance networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 254–265. ACM, 2011.

[24] A. Dovier, C. Piazza, and A. Policriti. A fast bisimulation algorithm. In *International Conference on*

Computer Aided Verification, pages 79–90. Springer, 2001.

[25] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 157–168. ACM, 2012.

[26] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *NSDI*, 2005.

[27] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ICFP*, 2011.

[28] J. Fu, P. Sjodin, and G. Karlsson. Loop-free updates of forwarding tables. *IEEE Transactions on Network and Service Management*, March 2008.

[29] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an operating system for networks. In *SIGCOMM CCR*, 2008.

[30] A. Guha, M. Reitblatt, and N. Foster. Machine-verified network controllers. *Programming Languages Design and Implementation*, 2013.

[31] B. Heller, C. Scott, N. McKeown, S. Shenker, A. Wundsam, H. Zeng, S. Whitlock, V. Jeyakumar, N. Handigol, M. McCauley, K. Zarifis, and P. Kazemian. Leveraging sdn layering to systematically troubleshoot networks. *HotSDN*, 2013.

[32] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. ElasticTree: Saving energy in data center networks. In *NSDI*, volume 3, pages 19–21, 2010.

[33] H. Hojjat, P. Reummer, J. McClurgh, P. Cerny, and N. Foster. Optimizing horn solvers for network repair. In *FMCAD*, 2016.

[34] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. *ACM SIGCOMM*, 2013.

[35] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined WAN. In *ACM SIGCOMM*, pages 3–14. ACM, 2013.

[36] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic scheduling of network updates. In *ACM SIGCOMM*, 2014.

[37] X. Jin, R. Mahajan, H. H. Liu, R. Gandhi, S. Kandula, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic scheduling of network updates. In *SIGCOMM*, 2014.

[38] N. P. Katta, J. Rexford, and D. Walker. Incremental consistent updates. *HotSDN*, 2013.

[39] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *NSDI*, 2013.

[40] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

[41] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

[42] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *HotSDN*, 2012.

[43] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *NSDI*, 2013.

[44] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. Kinetic: Verifiable dynamic network control. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, 2015.

[45] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. zUpdate: Updating data center networks with zero loss. *ACM SIGCOMM*, 2013.

[46] R. Mahajan and R. Wattenhofer. On consistent updates in software defined networks. *HotNets*, 2013.

[47] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *SIGCOMM*, 2011.

[48] M.Casado, T.Koponen, S.Shenker, and A.Tootoonchian. Fabric: A retrospective on evolving sdn. In *HotSDN*, 2012.

[49] J. McClurg, H. Hojjat, P. Cerny, and N. Foster. Efficient synthesis of network updates. *Programming*

Languages Design and Implementation, 2015.

[50] R. McGeer. A safe, efficient update protocol for OpenFlow networks. In *HotSDN*, 2012.

[51] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, and S. Shenker. OpenFlow: Enabling innovation in campus networks. In *SIGCOMM CCR*, 2008.

[52] S. Natarajan, X. Huang, and T. Wolf. Efficient conflict detection in flow-based virtualized networks. In *ICNC*, 2012.

[53] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The margrave tool for firewall analysis. In *LISA*, 2010.

[54] A. Noyes, T. Warszawski, P. Černỳ, and N. Foster. Toward synthesis of network updates. *SYNT*, 2014.

[55] Open Network Foundation. OpenFlow switch specification v1.4, October 2013. https://www.opennetworking.org/.

[56] P. Perešíni, M. Kuzniar, N. Vasic, M. Canini, and D. Kostic. OF.CPP: Consistent packet processing for OpenFlow. In *HotSDN*, 2013.

[57] G. D. Plotkin, N. Bjrner, N. P. Lopes, A. Rybalchenko, and G. Varghese. Scaling network verification using symmetry and surgery. In *POPL*, 2016.

[58] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A security enforcement kernel for OpenFlow networks. In *HotSDN*, 2012.

[59] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang. PGA: Using graphs to express and automatically reconcile network policies. In *SIGCOMM*, 2015.

[60] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. Simple-fying middlebox policy enforcement using sdn. In *SIGCOMM*, 2013.

[61] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker. Modular sdn programming with pyretic. In *USENIX ;login, 38(5)*, pages 40–47, October 2013.

[62] M. Reitblatt, M. Canini, A. Guha, and N. Foster. Fattire: Declarative fault tolerance for software-defined networks. 2013.

[63] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *SIGCOMM*, 2012.

[64] S. Saha, S. Prabhu, and P. Madhusudan. NetGen: Synthesizing data-plane configurations for network policies. In *SOSR*, 2015.

[65] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the production network be the testbed? In *OSDI*, 2010.

[66] L. Shi, J. Fu, and X. Fu. Loop-free forwarding table updates with minimal link overflow. *International Conference on Communications*, 2009.

[67] R. Soule, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A language for provisioning network resources. In *CoNEXT*, 2014.

[68] G. Varghese. Network Algorithmics: An interdisciplinary approach to designing fast networked devices, 2004.

[69] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying sdn programming using algorithmic policies. In *SIGCOMM*, 2013.

[70] G. Xie, J. Zhan, D. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of IP networks. In *INFOCOM*, 2005.

[71] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra. FIREMAN: A toolkit for firewall modeling and analysis. In *SnP*, 2006.

[72] W. Zhou, D. Jin, J. Croft, M. Caesar, and P. B. Godfrey. Enforcing customizable consistency properties in software-defined networks. In *NSDI*, 2015.